

Compiler 2

6. Block: Partielle Redundanz und ihre Entfernung



TECHNISCHE
UNIVERSITÄT
DARMSTADT





Einleitung



- ▶ Finde wiederholte Berechnungen auf einem Ausführungspfad
- ▶ ... und beseitige alle außer der ersten
- ▶ Bisher kennengelernte Verfahren
 - ▶ LVN, SVN, DVN
 - ▶ GCSE

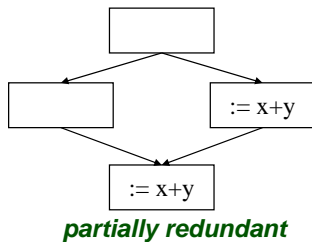
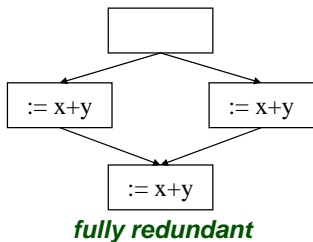
Totale Redundanz

Ein Ausdruck ist **total** redundant, wenn er auf **allen** Pfaden zu einer redundanten Verwendung berechnet wird.

Partielle Redundanz

Ein Ausdruck ist **partiell** redundant, wenn er auf **einigen, aber nicht auf allen** Pfaden zu einer redundanten Verwendung berechnet wird.

Totale und partielle Redundanz 2



CSE erkennt nur **totale** Redundanz.

Bewegung von schleifeninvarianten Anweisungen

- ▶ Loop Invariant Code Motion
- ▶ Bewege Anweisungen, die jede Iteration denselben Wert liefern
 - ▶ Dafür müssen die Operanden **schleifeninvariant** sein
- ▶ ... aus der Schleife **heraus**

Ein Operand ist **schleifeninvariant**, wenn

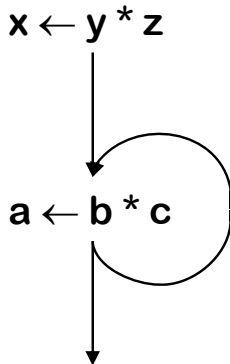
- ▶ er **konstant** ist, oder
- ▶ alle seine Definitionen **außerhalb** der Schleife liegen, oder
 - ▶ Erinnerung: Datenflußproblem REACHES
“Erreichende Definitionen”
- ▶ er eine einzelne Definition **innerhalb** der Schleife hat, die aber selbst invariant ist.

Naiver Ansatz: Bewege invariante Anweisungen **S** vor Schleifenkopf (pre-header).

Geht aber nur, wenn jedes **S**

- ▶ alle **Verwendungen** seiner LHS dominiert
- ▶ alle **Schleifenausgänge** dominiert
(break, continue, ...)

Schleifeninvariante Ausdrücke
sind eine Art der partiellen
Redundanz

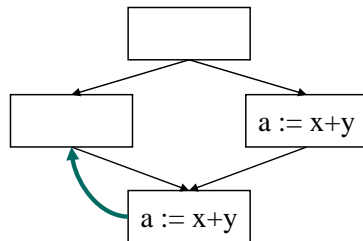


b*c nur redundant auf Rückwärtskante.

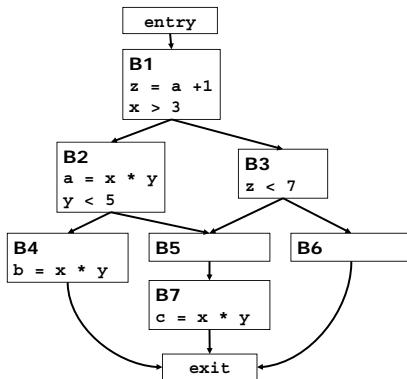


Eliminierung partieller Redundanz

- ▶ Bewege partiell redundante Berechnungen an ihre **optimalen** Stellen
 - ▶ Vermeide so Doppelberechnungen
- ▶ Beinhaltet CSE und LICM

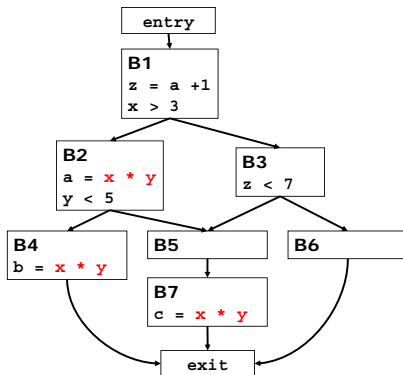


Beispiel



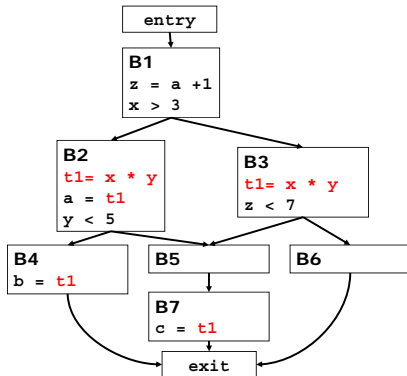
Welche Ausdrücke sind partiell redundant?

Beispiel

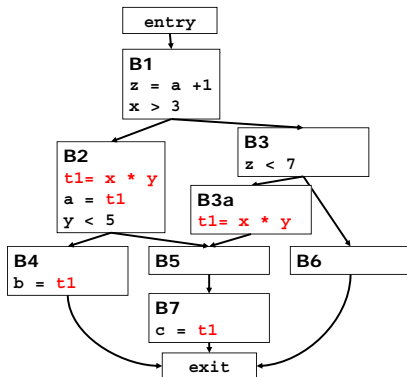


Wo die Berechnung tatsächlich durchführen, wo die Kopien verwenden?

Beispiel

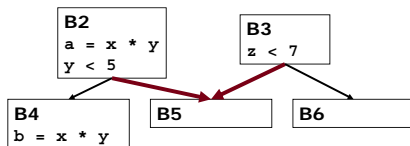


Ist das die optimale Lösung?

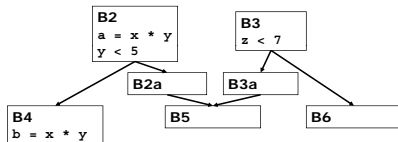


Was ist besonders an Kante **(B3, B5)**?

Vor Aufteilen

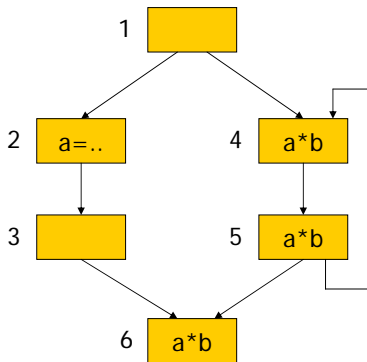


Alle kritischen Kanten aufgeteilt



Nachteil: Potentiell langsamer (Compile- und Laufzeit!)

- ▶ CSE: $a*b$ in **B5**
- ▶ LICM: $a*b$ in **B4**
- ▶ Bewegung: $a*b$ von **B6** nach **B3**



Optimale Lebenszeit

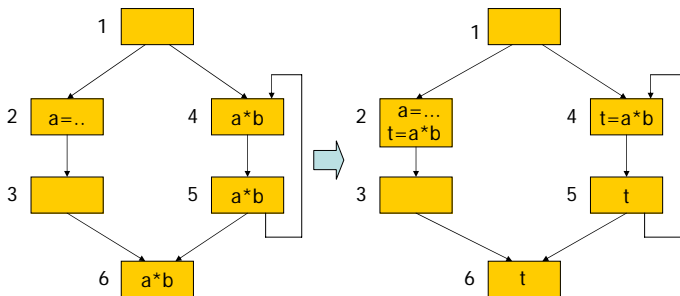
Die Lebenszeit von einer Neuberechnung zu einer Verwendung sollte so **kurz** wie möglich sein.

➔ Benötigt weniger Register

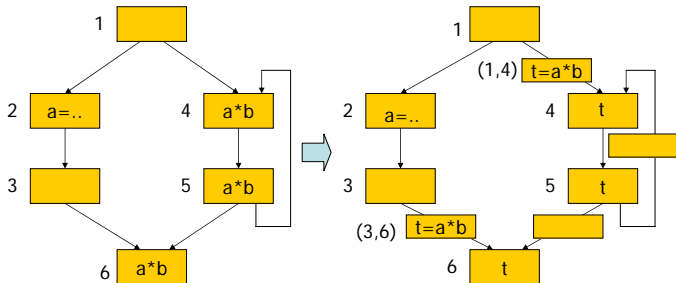
Berechnungsoptimalität

Zur Programmlaufzeit sollen so **wenige** Berechnungen wie möglich ausgeführt werden.

Algorithmus von Morel und Renvoise 1979



- ▶ Neue Berechnung von $a * b$ eingefügt in **B2**
- ▶ **B3** wäre besser (kürzere Lebenszeit!)
- ▶ $a * b$ in **B4** nicht berechnungsoptimal
 - ▶ Wegen (**B1**, **B2**) nicht am Ende von **B1** eingefügt
- ▶ $a * b$ gesichert in **B2+B4**, wiederverwendet in **B5+B6**



- ▶ Alle Kanten zu Join-Knoten aufteilen
- ▶ $a*b$ einfügen in **(B3,B6)** (hat optimale Lebenszeit!)
- ▶ $a*b$ einfügen in **(B1,B4)** (berechnungsoptimal)
- ▶ Erzeugt zusätzliche Blöcke (leere entfernbar)



Konzepte

Verfügbarkeit (Availability)

Ein Ausdruck e ist an einer Programmstelle p verfügbar, wenn sein Wert auf **allen** Pfaden von Programmanfang zu p berechnet wird.

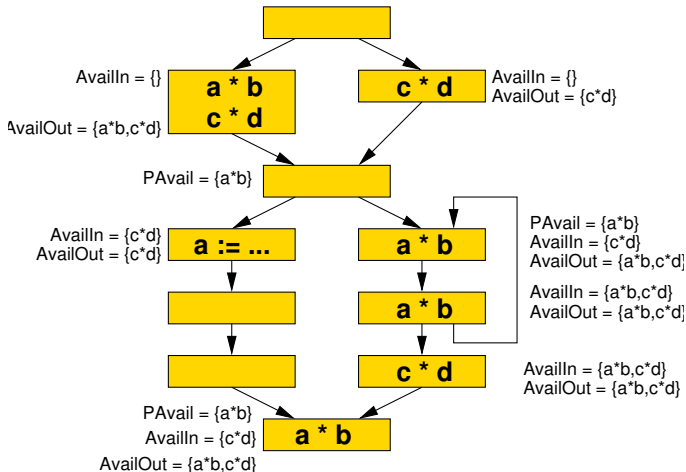
➔ Totale Redundanz von e an p

Partielle Verfügbarkeit (Partial Availability)

Ein Ausdruck e ist an einer Programmstelle p partiell verfügbar, wenn sein Wert auf **einigen** Pfaden von Programmanfang zu p berechnet wird.

➔ Partielle Redundanz von e an p

Beispiel: Verfügbare Ausdrücke



Nach oben exponierte Ausdrücke (Upwards Exposed)

Ausdrücke, deren Operanden vom Blockanfang bis zu ihrer Stelle nicht überschrieben werden sind nach **oben exponiert**.

↳ Ihre Berechnung könnte an den Blockanfang vorgezogen werden

- ▶ → UEEExpr
- ▶ Auch genannt **lokal vorziehbar** (locally anticipatable, ANTloc)

Nach unten exponierte Ausdrücke (Downward Exposed)

Ausdrücke, deren Operanden von ihrer Stelle bis zum Blockende nicht überschrieben werden sind nach **unten exponiert**.

➡ Ihre Berechnung könnte an das Blockende verzögert werden

- ▶ → DEExpr
- ▶ Heisst auch: Ausdrücke sind **lokal verfügbar**

Beispiel: Lokal exponierte Ausdrücke

a := 42

w := a + b

nicht UExpr/ANTloc, DEExpr

x := c + d

UExpr/ANTloc, DEExpr

y := e + f

UExpr/ANTloc, nicht DEExpr

z := a + e

nicht UExpr/ANTloc, nicht DEExpr

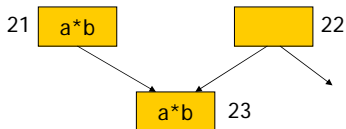
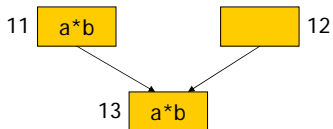
e := 23

Vorziehbarkeit (Anticipatability)

Die Berechnung eines Ausdrucks e ist **vorziehbar** an eine Programmstelle p , wenn er auf allen Pfaden von p zum Programmende berechnet wird.

➡ Auch genannt: Very Busy Expression

Beispiel: Vorziehbarkeit



- ▶ $a*b$ vorziehbar nach Block 12
- ▶ Aber **nicht** nach Block 22

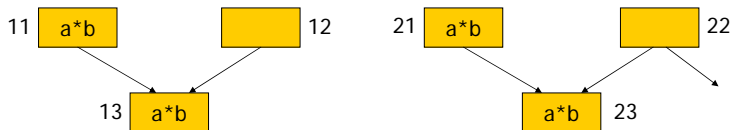
Sichere Berechnung

Ein Ausdruck e kann an der Stelle p **sicher** berechnet werden, wenn er dort bereits **verfügbar** ist oder dorthin **vorgezogen** werden kann.

➡ Ziel: Gleicher Wert ohne weitere eventuelle Berechnungsfehler (exceptions, z.B. Division-by-Zero)

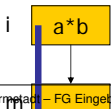
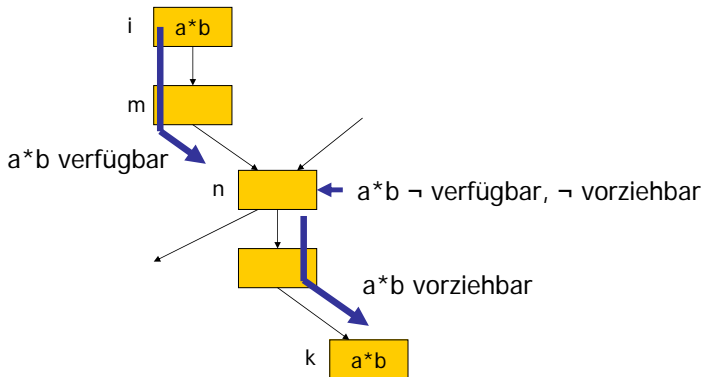
- ▶ Im 1. Fall wurde der Ausdruck so bereits berechnet und könnte gefahrlos noch ein weiteres Mal berechnet werden
- ▶ Im 2. Fall würden eventuelle Fehler ohnehin auftreten, da dieser Ausdruck später in jedem Fall berechnet würde

Beispiel: Sicherheit



- ▶ Neuberechnung von $a*b$ in Block 12 ist sicher
- ▶ $a*b$ in Block 22 ist unsicher
- ▶ $a*b$ in **Kante** (Block 22, Block 23) wäre sicher

Beispiel: Verfügbar, vorziehbar, sicher



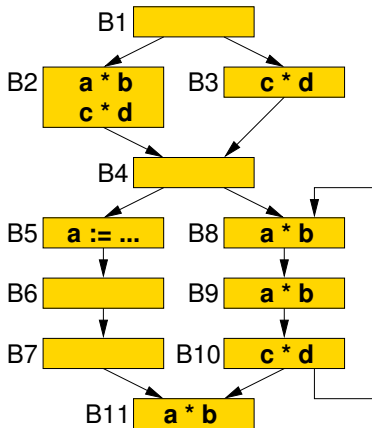
Transparenz

Ein Block **B** ist **transparent** in Hinblick auf einen Ausdruck e , wenn er selbst keine Zuweisungen an Operanden von e enthält.

Leere

Ein Block **B** ist **leer** in Hinblick auf einen Ausdruck e , wenn er selbst weder eine Auswertung von e enthält noch Zuweisungen an Operanden von e .

➔ Schreibweise: $\text{empty}(\mathbf{B}) = \text{TRUE}$ bezüglich e



► Für $a * b$

- Alle Blöcke ausser **B5** sind transparent
- $b \in \{ \mathbf{B1, B3, B4, B6, B7, B10} \}$:
 $\text{empty}(b) = \text{TRUE}$

► Für $c * d$

- Alle Blöcke sind transparent
- $b \in \{ \mathbf{B1, B4, B5, B6, B7, B8, B9, B11} \}$:
 $\text{empty}(b) = \text{TRUE}$

- ▶ Bearbeite **alle** partiell redundanten Verwendungen von Ausdrücken e
 - ▶ Berechnungsoptimalität
- ▶ Füge Neuberechnungen von e an **sicheren** Stellen ein
- ▶ Lösche nun total redundant gewordene Berechnungen von e
- ▶ Achte auf **kurze Lebenszeiten** von Neuberechneten Werten
- ▶ Vermeide unnötiges Aufteilen von Kanten
- ▶ Bevorzuge schnelleren Algorithmus
- ▶ Am besten auch noch möglichst einfach zu verstehen



PRE mit Eliminierungspfaden



E-path_PRE – Partial Redundancy Elimination Made Easy

ACM SIGPLAN Notices, 2002, vol. 37, no 8, pp. 53-65

- ▶ Dhanajay M. Dhamdhere
- ▶ Korrigierte Fassung auf OC Web-Site
- ▶ Verfeinert zusammen mit Dheeraj Kumar 2006



Eliminierbarkeitspfad (E-Pfad, eliminatability path)

Ein E-Pfad für einen Ausdruck e ist ein Pfad b_i, \dots, b_k im CFG so dass

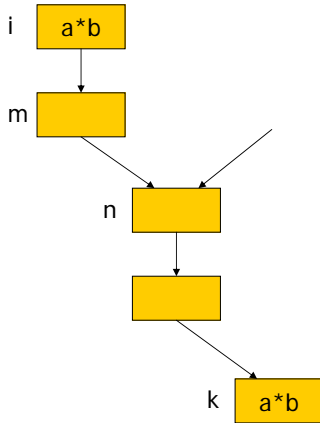
1. e lokal **verfügbar** in b_i und lokal **vorziehbar** in b_k ist
2. Für $b \in (b_i, \dots, b_k)$ gilt: **empty**(b) = TRUE
3. e ist **sicher** auf jeder Ausgangskante eines Blocks $b \in [b_i, \dots, b_k)$

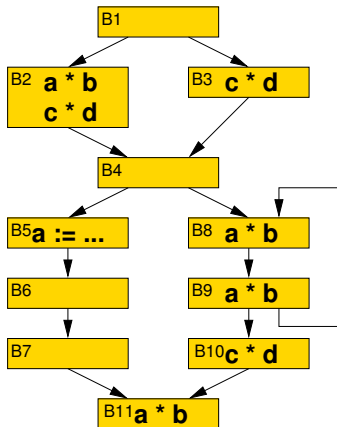
Notation: Ein Pfad $[b_i, \dots, b_k]$ enthält seine Anfangs- und Endblöcke, ein Pfad (b_i, \dots, b_k) nicht.

Beispiel: E-Pfad



- ▶ $a*b$ ist **verfügbar** am Ende von $[j \dots m]$
- ▶ $a*b$ ist **vorziehbar** am Ende von $[n \dots k]$
- ▶ Berechnung von $a*b$ in Block k ist **eliminierbar**





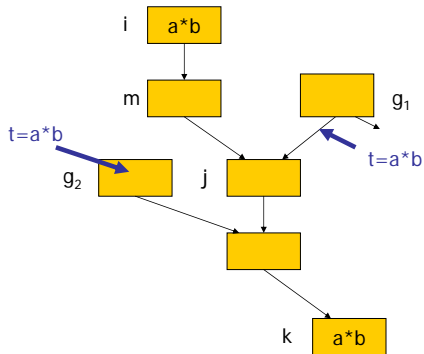
► Für $a*b$

- $[b_8, b_9]$
- $[b_9, b_8]$
- $[b_9, b_{10}, b_{11}]$
- **Nicht:** $[b_2, b_4, b_8]$

► Für $c*d$

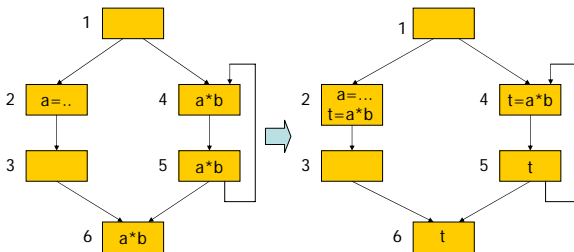
- $[b_2, b_4, b_8, b_9, b_{10}]$
- $[b_3, b_4, b_8, b_9, b_{10}]$

- ▶ Das erste Vorkommen von e im Endblock b_k des E-Pfades $[b_i, \dots, b_k]$ ist **eliminierbar**
- ▶ Das vorherige Evaluationsergebnis des Ausdrucks e wird dazu **gesichert** in der Variable t_e
- ▶ Wird der Pfad $(b_i, \dots, b_k]$ von einem Block b_h **ausserhalb** über eine Kante (b_h, b_j) betreten
- ▶ ... muss e (falls nötig) mit $t_e := e$ **berechnet** und **gesichert** werden ...
 - ▶ Am Ende des **Blocks** b_h , falls $|\text{succ}(b_h)| = 1$
 - ▶ Auf der aufgeteilten **Kante** (b_h, b_j) sonst
 - ▶ Lebenszeitoptimale Platzierung (so spät wie möglich)
- ▶ Eine Neuberechnung und Sicherung ist nicht nötig, falls b_h **selber** auf einem E-Pfad für e liegt.



- ▶ E-Pfad: $[j, \dots, k]$
- ▶ Einfügen von Berechnungen von e und Kopieren
 - ▶ In Kante (g_1, j) und Block g_2

Vergleich mit Ergebnis nach MRA



- ▶ E-Pfade: [4, 5], [5, 4], [5, 6]
- ▶ Herstellen totaler Redundanz: Einfügen in Ast 2,3
 - ▶ Nach MRA in 2, mit E-Pfaden in 3: Lebenszeitoptimal
- ▶ Entfernen von Redundanz in 4
 - ▶ Nach MRA nicht möglich
 - ▶ [5, 4] ist E-Pfad, Berechnung in 4 muss entfernbare sein
→ Kante (1,4) aufteilen, dort Berechnen und Kopieren

Wert von e sichern

Füge Anweisung $t_e := e$ vor einer Benutzung von e ein und ersetze Benutzung durch t_e

Neue Berechnung von e einfügen

Füge Anweisung $t_e := e$ ein

Eliminiere redundante Berechnung von e

Ersetze e durch t_e

An welchen Stellen diese Aktionen ausführen?



Neue Notation

Prädikate bestimmen für jeden Ausdruck e , ob eine Aussage wahr oder falsch ist

Wahr für alle am Ende eines Blocks b verfügbaren Ausdr.

$\text{avail}(b) = \text{TRUE}$ für $e \Leftrightarrow e \in \text{AVAILOUT}(b)$

Wahr für alle in den Block b vorziehbaren Ausdrücke

$\text{ant}(b) = \text{TRUE}$ für $e \Leftrightarrow e \in \text{ANT}(b)$

Wahr für alle Ausdrücke, für die beides gilt

$\text{avail}(b) \cdot \text{ant}(b) = \text{TRUE} \Leftrightarrow e \in (\text{AVAILOUT}(b) \cap \text{ANT}(b))$

Wahr für alle Ausdrücke, für die mindestens eines gilt

$\text{avail}(b) + \text{ant}(b) = \text{TRUE} \Leftrightarrow e \in (\text{AVAILOUT}(b) \cup \text{ANT}(b))$



Konjunktion

$cpin(b) = \prod_p cpout(p) = \text{TRUE}$ für Kopie $c \Leftrightarrow c \in \bigcap_{p \in pred(b)} CPOUT(p)$

Disjunktion

$liveout(b) = \sum_s livein(s) = \text{TRUE}$ für Variable $v \Leftrightarrow v \in \bigcup_{s \in succ(b)} LIVEIN(s)$

Für alle Ausdrücke eindeutige Zuordnung festlegen

...

a*b → Bit 1
c*d → Bit 0

Wert [1 : 0]	entspricht Prädikat x	entspricht Menge
00	$\neg X_{a*b}, \neg X_{c*d}$	\emptyset
01	$\neg X_{a*b}, X_{c*d}$	$\{c * d\}$
10	$X_{a*b}, \neg X_{c*d}$	$\{a * b\}$
11	X_{a*b}, X_{c*d}	$\{a * b, c * d\}$

- ▶ Bitvektoren können sehr lang werden
- ▶ Prädikate liefern Ergebnisse als Bitvektor
- ▶ Damit logische Verknüpfungen effizient implementierbar

Beispiel: Verküpfung von Prädikaten

Zuordnung von Ausdrücken e an Bits

$a*b \rightarrow$ Bit 0

$c*d \rightarrow$ Bit 1

$a*d \rightarrow$ Bit 2

...

Gegeben: Zwei Prädikate x und y bezüglich Ausdruck e

Annahme: $x = 101, y = 011$

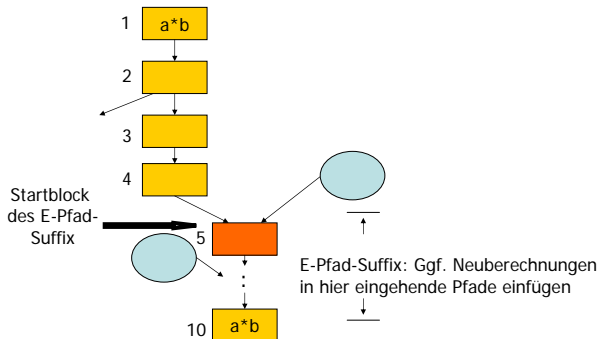
$$x \cdot y = x \& y = 001$$

$$x + y = x | y = 111$$



Bestimmen von E-Pfaden

Anatomie eines E-Pfades

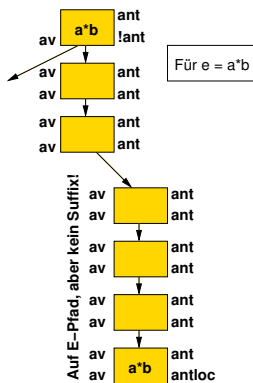


$$\begin{aligned} b \in (1, 2] & \quad \text{avail}(b) \cdot \neg \text{ant}(b) \\ b \in [3, 4] & \quad \text{avail}(b) \cdot \text{ant}(b) \\ b \in [5, \dots, 10] & \quad \neg \text{avail}(b) \cdot \text{ant}(b) \end{aligned}$$

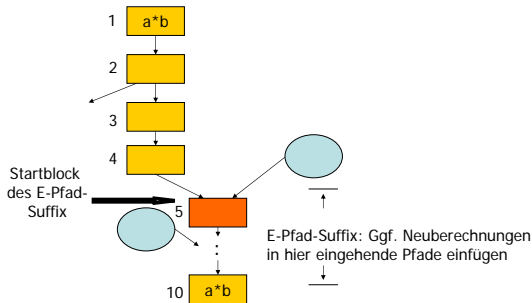
E-Pfad-Suffix

Beispiel: Leerer E-Pfad-Suffix

Anforderung an E-Pfad-Suffix: $\neg \text{avail}(b) \cdot \text{ant}(b)$



- ▶ E-Pfad $[b_i, \dots, b_k]$ hat drei (ggf. leere) Segmente
 - ▶ $\text{avail}(b) \cdot \neg \text{ant}(b)$
 - ▶ $\text{avail}(b) \cdot \text{ant}(b)$
 - ▶ $\neg \text{avail}(b) \cdot \text{ant}(b)$
- ▶ Finde **Start des E-Pfad-Suffixes**: Suche Block b_m mit $\neg \text{avail}(b_m) \cdot \text{ant}(b_m) \cdot \sum_p \text{avail}(p)$
- ▶ Suche rückwärts von b_m durch Segmente
 - $\text{avail}(b) \cdot \text{ant}(b)$
 - $\text{avail}(b) \cdot \neg \text{ant}(b)$bis **Start des E-Pfades** b_i erreicht
- ▶ Suche vorwärts von b_m durch Segment
 - $\neg \text{avail}(b) \cdot \text{ant}(b)$bis **Ende des E-Pfades** erreicht



Start des E-Pfades Sichere Wert von e in t_e

E-Pfad-Suffix Füge Neuberechnungen $t_e := e$ in eingehende Pfade ein

Ende des E-Pfades Ersetze redundante Berechnung von e durch t_e

- ▶ Löse E-Pfad-Problem für alle Ausdrücke **gleichzeitig**
- ▶ Erinnerung: Ergebnisse der Prädikate sind Bitvektoren
- ▶ Bitweise Verknüpfung mit AND und OR
- ▶ Berechnet nicht nur E-Pfade
- ▶ Direkte Bestimmung von
 - ▶ Blöcken mit zu eliminierenden Ausdrücken
 - ▶ Einfügestellen (Blöcke, Kanten) für Neuberechnungen
 - ▶ Stellen für Sichern von Werten



Datenflussproblem



- ▶ Datenflussproblem über Prädikate von **Ausdrücken**
- ▶ Zerlegung in
 - ▶ Verschiedene Teilprobleme
 - ▶ Vorberechenbare Eigenschaften
 - ▶ Herleitbare Eigenschaften (durch DF-Löser)

Ausdruck ist lokal verfügbar (locally available)

$\text{comp}(b) = \text{TRUE}$: e wird in b berechnet (computed) und seine Operanden hinterher nicht überschrieben

↳ $e \in \text{DEEXPR}(b)$

Ausdruck ist lokal vorziehbar (locally anticipatable)

$\text{antloc}(b) = \text{TRUE}$: Operanden von e werden vor Berechnung nicht zugewiesen

↳ $e \in \text{UEEXPR}(b)$

Block ist transparent für Ausdruck

$\text{transp}(b) = \text{TRUE}$: b hat keine Zuweisungen an Operanden von e

↳ $e \notin \text{EXPRKILL}(b)$

Beispiel für vorberechenbare Eigenschaften

Bit 1: a+b
Bit 2: c+d
Bit 3: e+f
Bit 4: a+e
Bit 5: q+r
Bit 6: x+y

comp(B1) = 110000
antloc(B1) = 011000
transp(B1) = 010010

B1

a := 42

w := a + b

x := c + d

y := e + f

z := a + e

e := 23

nicht UExpr/ANTloc, DEExpr

UExpr/ANTloc, DEExpr

UExpr/ANTloc, nicht DEExpr

nicht UExpr/ANTloc, nicht DEExpr



$availin(b)$ e ist verfügbar bei Eintritt in b

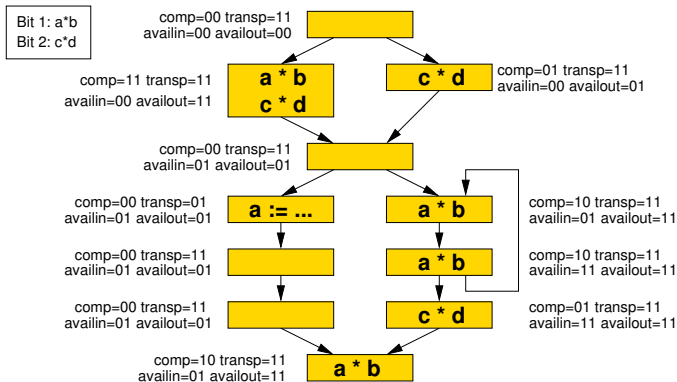
$availout(b)$ e ist verfügbar bei Austritt aus b

Berechnung

$$availin(b) = \prod_p availout(p)$$

$$availout(b) = availin(b) \cdot transp(b) + comp(b)$$

Beispiel verfügbare Ausdrücke



$$\begin{aligned}
 availin(b) &= \prod_p availout(p) \\
 availout(b) &= availin(b) \cdot transp(b) + comp(b)
 \end{aligned}$$

$\text{antin}(b)$ e ist vorziehbar an Blockanfang von b

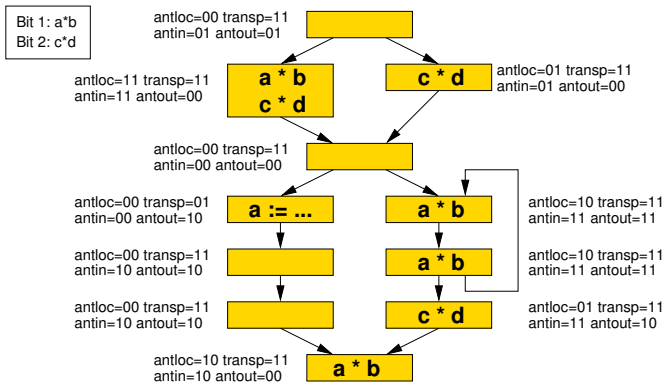
$\text{antout}(b)$ e ist vorziehbar an Blockende von b

Berechnung

$$\text{antin}(b) = \text{antout}(b) \cdot \text{transp}(b) + \text{antloc}(b)$$

$$\text{antout}(b) = \prod_s \text{antin}(s)$$

Beispiel vorziehbare Ausdrücke



$$\text{antin}(b) = \text{antout}(b) \cdot \text{transp}(b) + \text{antloc}(b)$$

$$\text{antout}(b) = \prod_s \text{antin}(s)$$

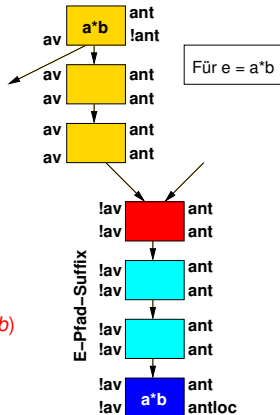
Bestimmen von Blöcken in E-Pfad-Suffix

$\text{epsin}(b)$ Blockanfang von b liegt auf einem E-Pfad-Suffix für e

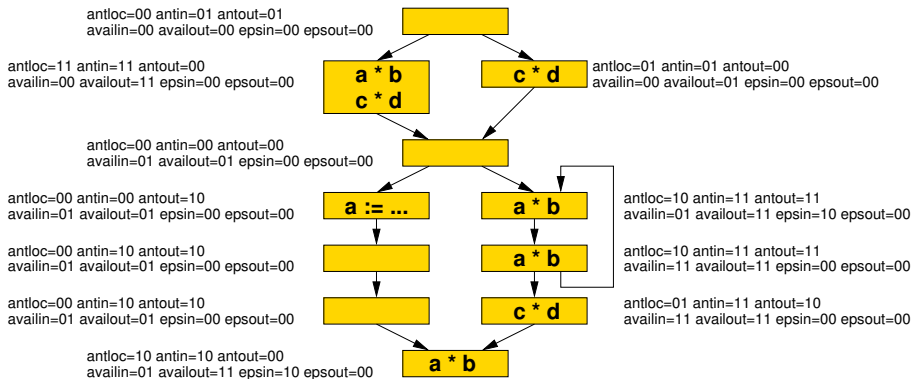
$\text{epsout}(b)$ Blockende von b liegt auf einem E-Pfad-Suffix für e

Berechnung

$$\begin{aligned}\text{epsin}(b) &= (\sum_p (\text{availout}(p) + \text{epsout}(p))) \cdot \text{antin}(b) \cdot \neg \text{availin}(b) \\ \text{epsout}(b) &= \text{epsin}(b) \cdot \neg \text{antloc}(b)\end{aligned}$$



Beispiel E-Pfad-Suffix



$$\begin{aligned} \text{epsin}(b) &= (\sum_p (\text{availout}(p) + \text{epsout}(p))) \cdot \text{antin}(b) \cdot \neg \text{availin}(b) \\ \text{epsout}(b) &= \text{epsin}(b) \cdot \neg \text{antloc}(b) \end{aligned}$$

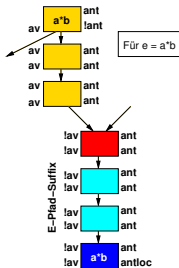
Idee: Bestimmen der Blöcke der redundanten Berechnungen

Redundante $e \in \text{UEExpr}(b_k)$ liegen im **Endblock** b_k eines E-Pfades

E-Pfad-Suffix $\neq \emptyset$

Endblock des **E-Pfad-Suffixes**

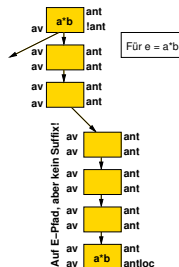
$\Leftrightarrow \text{epsin}(b) \cdot \text{antloc}(b)$



E-Pfad-Suffix = \emptyset

Endblock des **E-Pfades**

$\Leftrightarrow \text{availin}(b) \cdot \text{antloc}(b)$





redund(b) Berechnung von e in b ist redundant und kann durch τ_e ersetzt werden

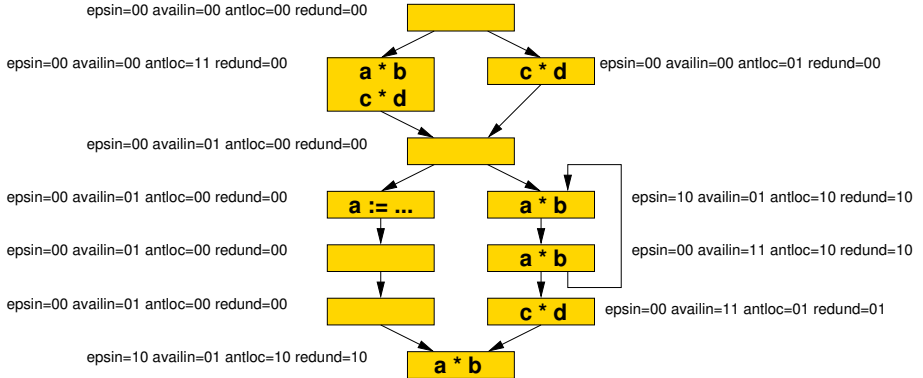
Berechnung

$$\text{redund}(b) = (\text{epsin}(b) + \text{availin}(b)) \cdot \text{antloc}(b)$$

Genauer: Alle lokal antizipierbaren e , also $e \in \text{UEExpr}(b_k)$ sind redundant.

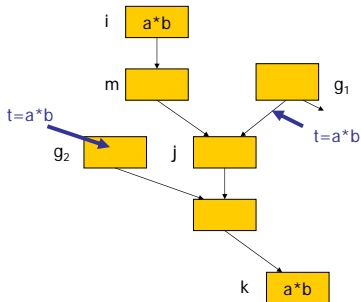
↳ Können **eliminiert** werden

Beispiel Redundanz



$$\text{redund}(b) = (\text{epsin}(b) + \text{availin}(b)) \cdot \text{antloc}(b)$$

An Eintrittspunkten in den Pfad!



- ▶ Wenn alle Nachfolger von externem Block in E-Pfad:
In **Block**
- ▶ Sonst in aufgeteilte **Kante**

Einfügestellen für neue Berechnungen bestimmen

$\text{insert}(b_h)$ Füge $t_e := e$ am Ende von Block b_h ein

$\text{insert}(b_h, b_j)$ Füge $t_e := e$ in aufgeteilte Kante (b_h, b_j) ein

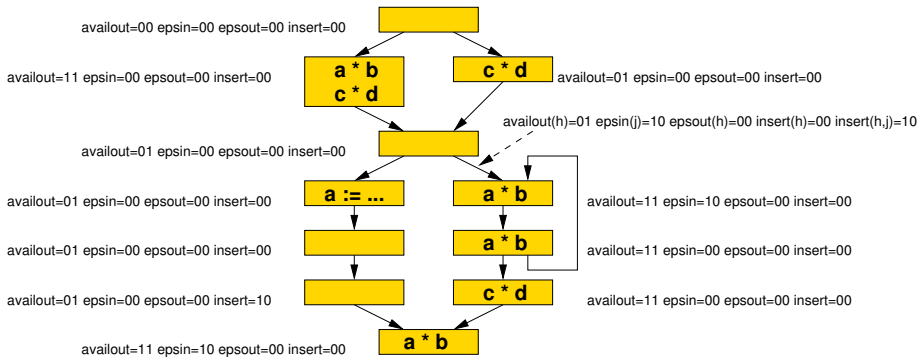
$b_h \notin E\text{-Pfad}$, $b_j \in E\text{-Pfad } (b_i, \dots, b_k]$

Berechnung

$$\text{insert}(b_h) = \neg \text{availout}(b_h) \cdot \neg \text{epsout}(b_h) \cdot \prod_s \text{epsin}(s)$$

$$\text{insert}(b_h, b_j) = \neg \text{availout}(b_h) \cdot \neg \text{epsout}(b_h) \cdot \neg \text{insert}(b_h) \cdot \text{epsin}(b_j)$$

Beispiel Einfügen von Berechnungen

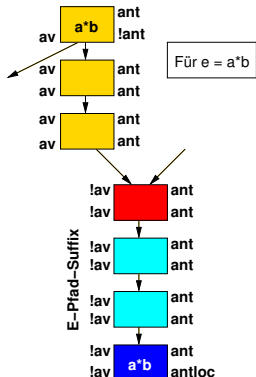


$$\text{insert}(b_h) = \neg \text{availout}(b_h) \cdot \neg \text{epsout}(b_h) \cdot \prod_s \text{epsin}(s)$$

$$\text{insert}(b_h, b_j) = \neg \text{availout}(b_h) \cdot \neg \text{epsout}(b_h) \cdot \neg \text{insert}(b_h) \cdot \text{epsin}(b_j)$$

Sichern von Berechnungsergebnissen

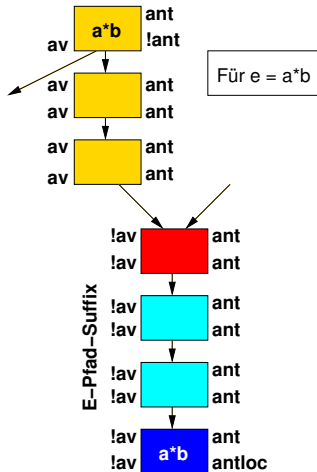
Am Anfang des E-Pfades!



➡ **Startblock** des E-Pfades bestimmen!

Idee zur Bestimmung des Startblocks

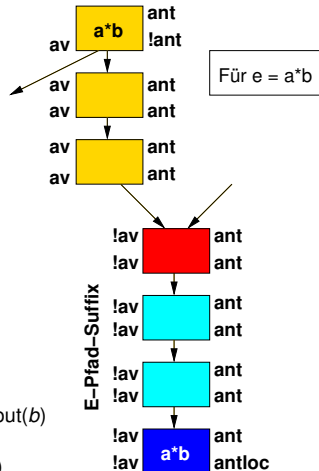
- ▶ Beginne bei bekanntem Block und suche **rückwärts**
- ▶ Beginne bei **Startblock des E-Pfad-Suffix**
 - ▶ ... falls E-Pfad einen Suffix hat
- ▶ Sonst: Suche von **Endblock des E-Pfades** aus rückwärts
- ▶ Bis nicht-redundante Berechnung von e gefunden



Berechnung des Startblocks



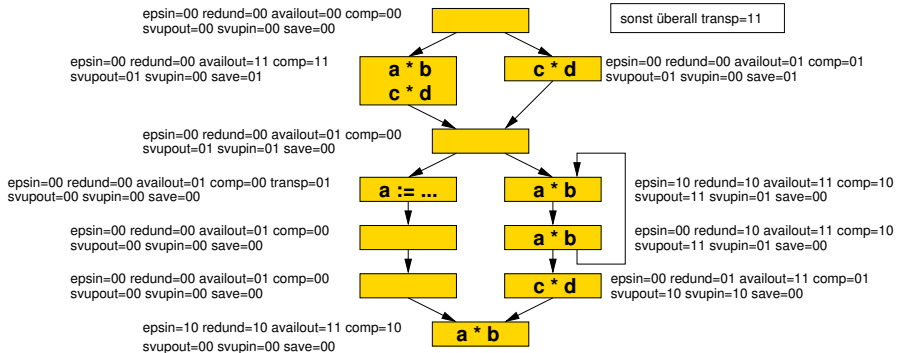
- svupin(b)** Sichere Ergebnis für e **in** Block vom Ausgang zum Eingang von b
- svupout(b)** Sichere Ergebnis für e **über** Blockgrenze zum Ausgang von b
- save(b)** Sichere Ergebnis in τ_e in Block b



Berechnung

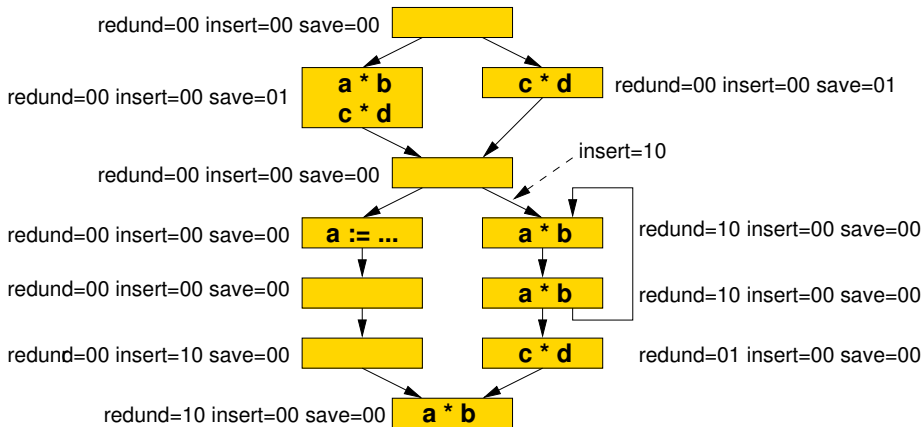
$$\begin{aligned}
 \text{svupout}(b) &= (\sum_s (\text{epsin}(s) + \text{redund}(s) + \text{svupin}(s))) \cdot \text{availout}(b) \\
 \text{svupin}(b) &= \text{svupout}(b) \cdot \neg \text{comp}(b) \\
 \text{save}(b) &= \text{svupout}(b) \cdot \text{comp}(b) \cdot \neg (\text{redund}(b) \cdot \text{transp}(b))
 \end{aligned}$$

Beispiel Bestimmung der E-Pfad-Startblöcke

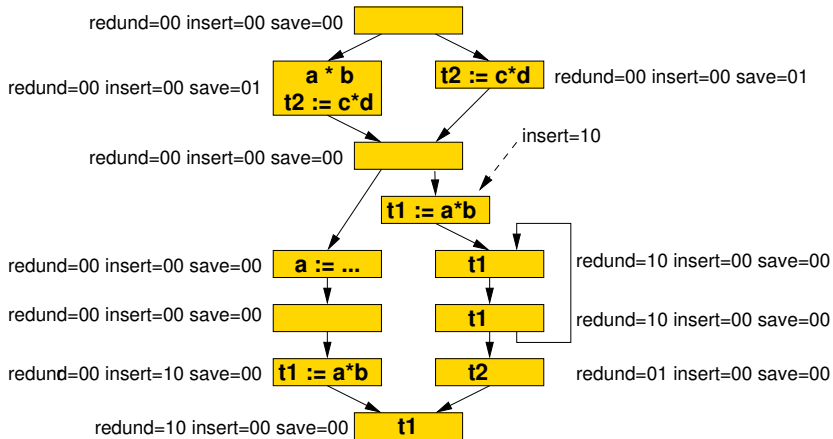


$$\begin{aligned} \text{svupout}(b) &= \left(\sum_s (\text{epsin}(s) + \text{redund}(s) + \text{svupin}(s)) \right) \cdot \text{availout}(b) \\ \text{svupin}(b) &= \text{svupout}(b) \cdot \neg \text{comp}(b) \\ \text{save}(b) &= \text{svupout}(b) \cdot \text{comp}(b) \cdot \neg (\text{redund}(b) \cdot \text{transp}(b)) \end{aligned}$$

Beispiel Zusammenfassung Datenflussanalyse



Beispiel PRE-Optimierung



- ▶ PRE ist eine sehr mächtige Optimierung
- ▶ Lösbar durch komplexes Datenflußproblem
 - ▶ Hier aber schon deutlich einfacher als klassische Verfahren!
- ▶ Noch weitere Verfeinerung möglich
 - ▶ Optimierung auf SSA-Form → SSA-PRE
 - ▶ Verfeinerte Stellung des Datenflussproblems
→ D. Kumar 2006, geringerer Rechenaufwand