Compiler 2

6. Block: Partielle Redundanz und ihre Entfernung

TECHNISCHE UNIVERSITÄT DARMSTADT

TECHNISCHE UNIVERSITÄT DARMSTADT

Einleitung

Common Subexpression Elimination

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Totale und partielle Redundanz 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Totale Redundanz

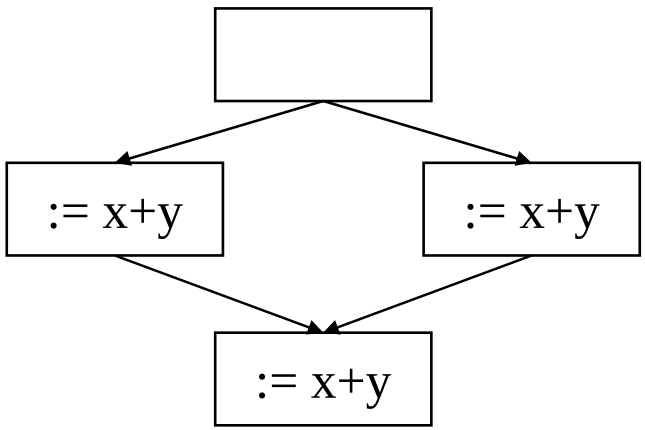
Ein Ausdruck ist total redundant, wenn er auf allen Pfaden zu einer redundanten Verwendung berechnet wird.

Partielle Redundanz

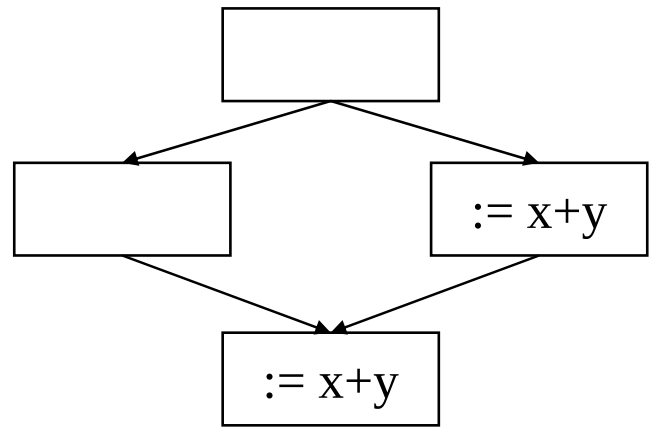
Ein Ausdruck ist partiell redundant, wenn er auf einigen, aber nicht auf allen Pfaden zu einer redundanten Verwendung berechnet wird.

Totale und partielle Redundanz 2

TECHNISCHE UNIVERSITÄT DARMSTADT



fully redundant



partially redundant

CSE erkennt nur totale Redundanz.

Quelle: CMU CS 15-745 2009

Bewegung von schleifeninvarianten Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Schleifeninvarianz

TECHNISCHE UNIVERSITÄT DARMSTADT

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.

Loop Invariant Code Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

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

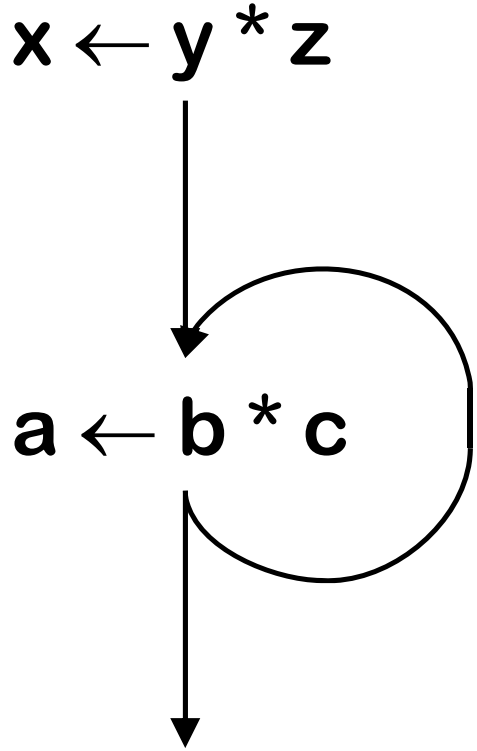
Geht aber nur, wenn jedes **S**

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

Zusammenhang LICM und partielle Redundanz

TECHNISCHE UNIVERSITÄT DARMSTADT

Schleifeninvariante Ausdrücke sind eine Art der partiellen Redundanz



b*c nur redundant auf Rückwärtskante.

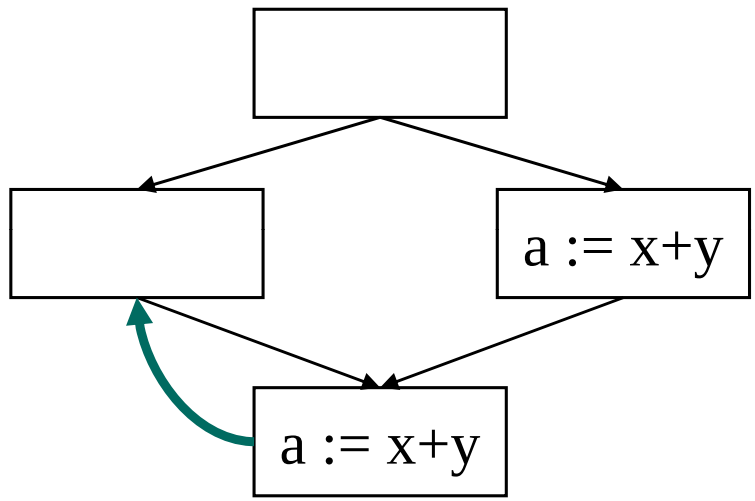
Quelle: CMU CS 15-745 2009

Eliminierung partieller Redundanz

Eliminierung partieller Redundanz: Idee

TECHNISCHE UNIVERSITÄT DARMSTADT

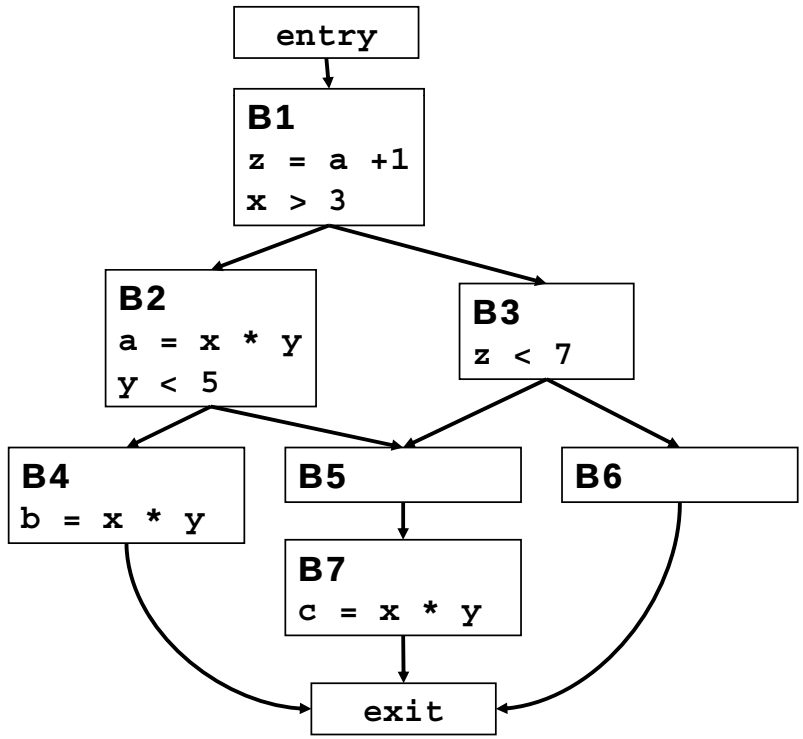
- Bewege partiell redundante Berechnungen an ihre optimalen Stellen
  - Vermeide so Doppelberechnungen
- Beinhaltet CSE und LICM



Quelle: CMU CS 15-745 2009

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

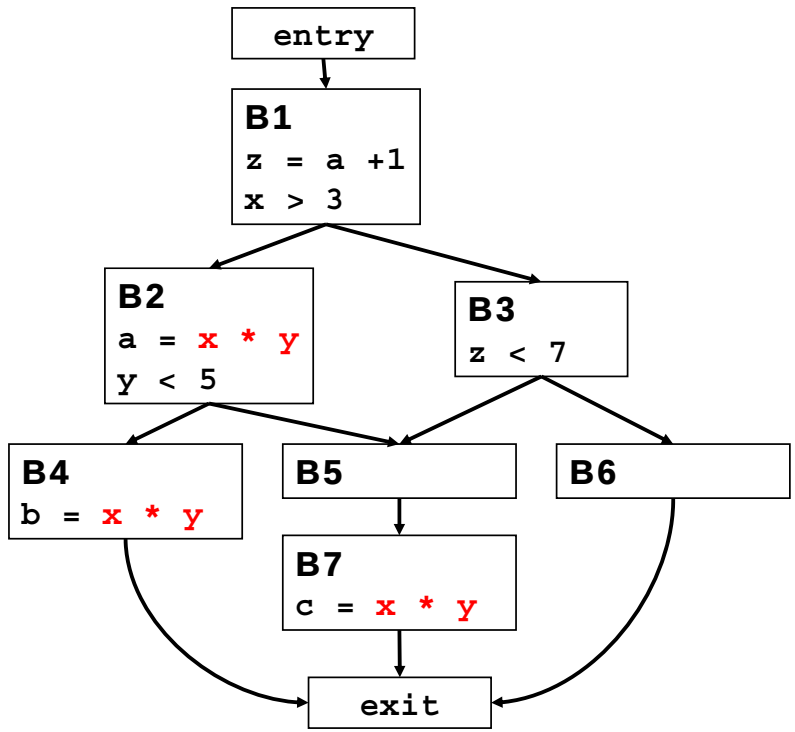


Welche Ausdrücke sind partiell redundant?

Quelle: CMU CS 15-745 2009

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

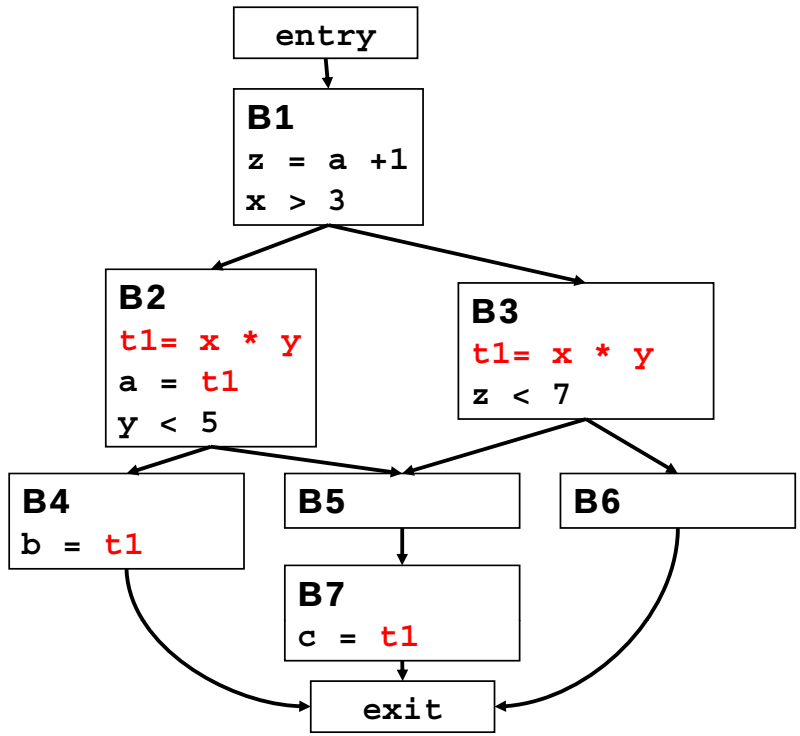


Wo die Berechnung tatsächlich durchführen, wo die Kopien verwenden?

Quelle: CMU CS 15-745 2009

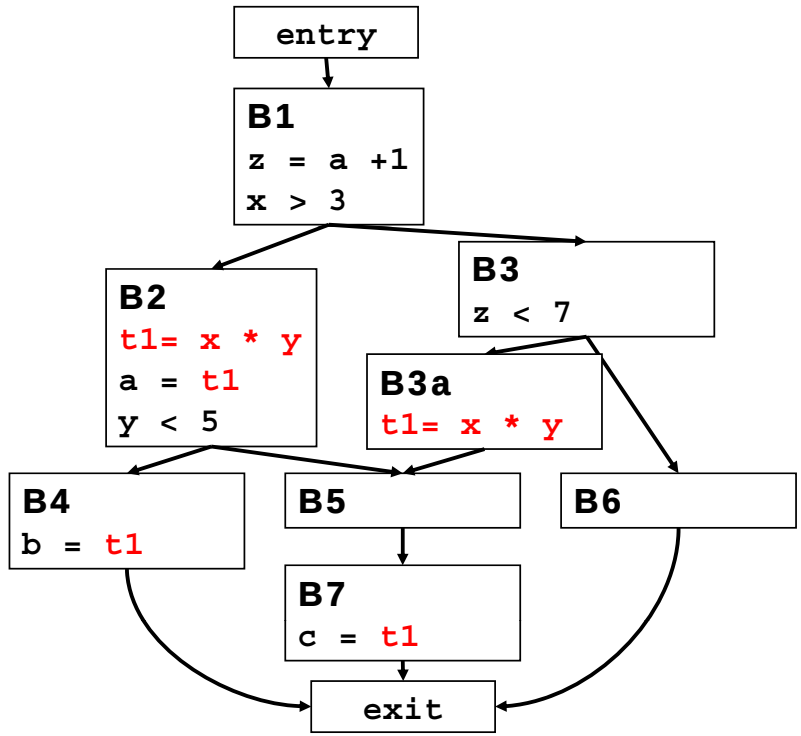
Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT



Ist das die optimale Lösung?

Beispiel



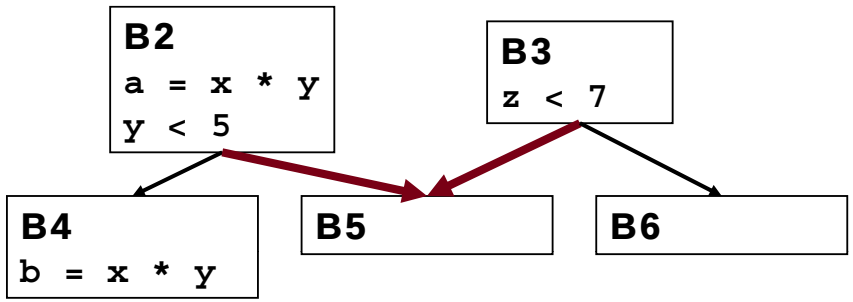
Was ist besonders an Kante **(B3, B5)**?

Quelle: CMU CS 15-745 2009

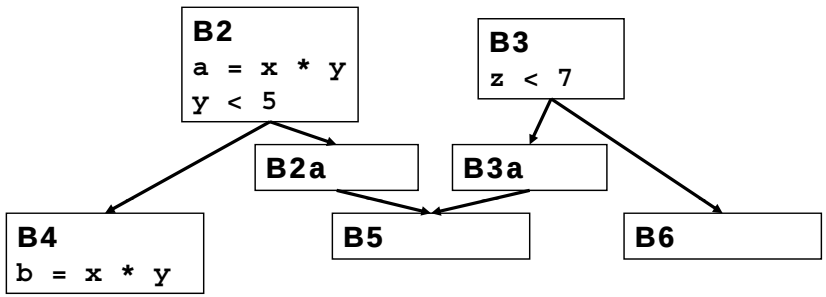
Kritische Kante

TECHNISCHE UNIVERSITÄT DARMSTADT

Vor Aufteilen



Alle kritischen Kanten aufgeteilt

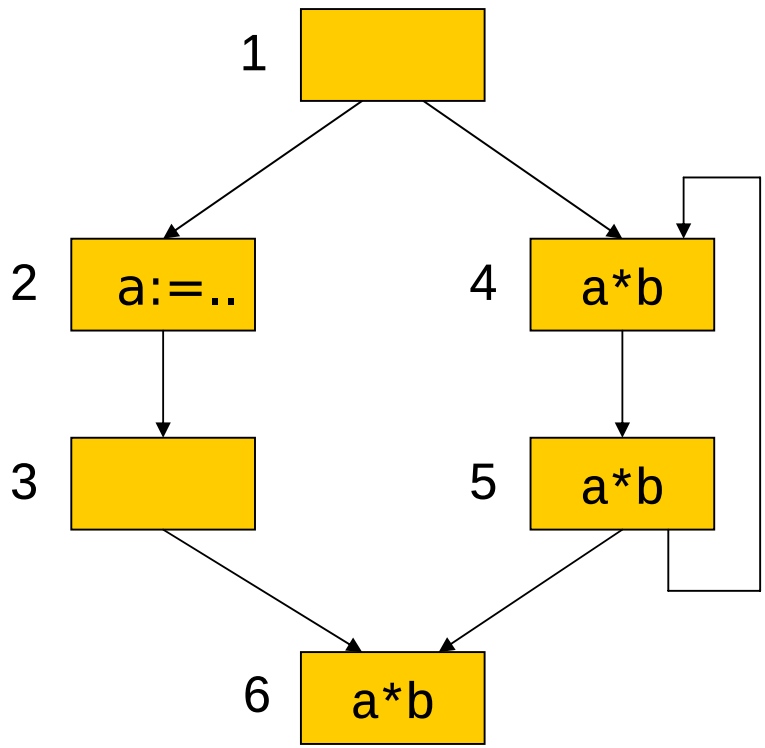


Nachteil: Potentiell langsamer (Compile- und Laufzeit!)

PRE kombiniert drei Optimierungen

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Quelle: Dhamdhere, Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Qualitätskriterien

TECHNISCHE UNIVERSITÄT DARMSTADT

Optimale Lebenszeit

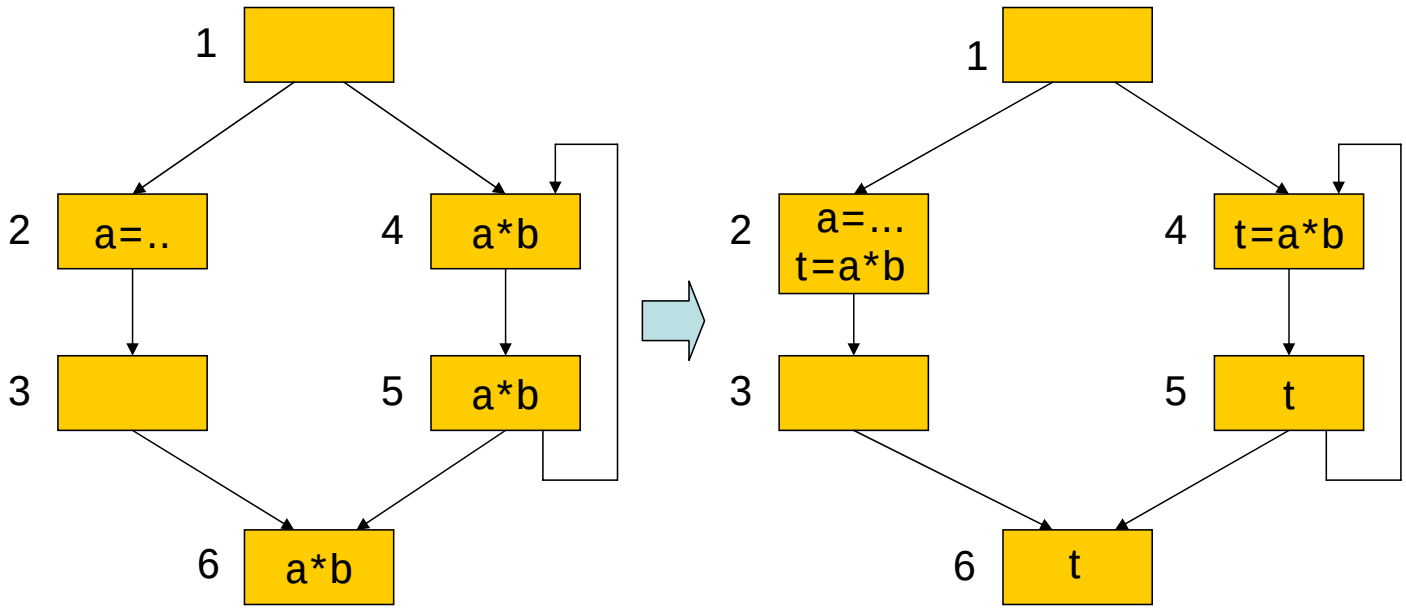
Die Lebenszeit von einer Neuberechnung zu einer Verwendung sollte so kurz wie möglich sein.

➥Benötigt weniger Register

Berechnungsoptimalität

Zur Programmlaufzeit sollen so wenige Berechnungen wie möglich ausgeführt werden.

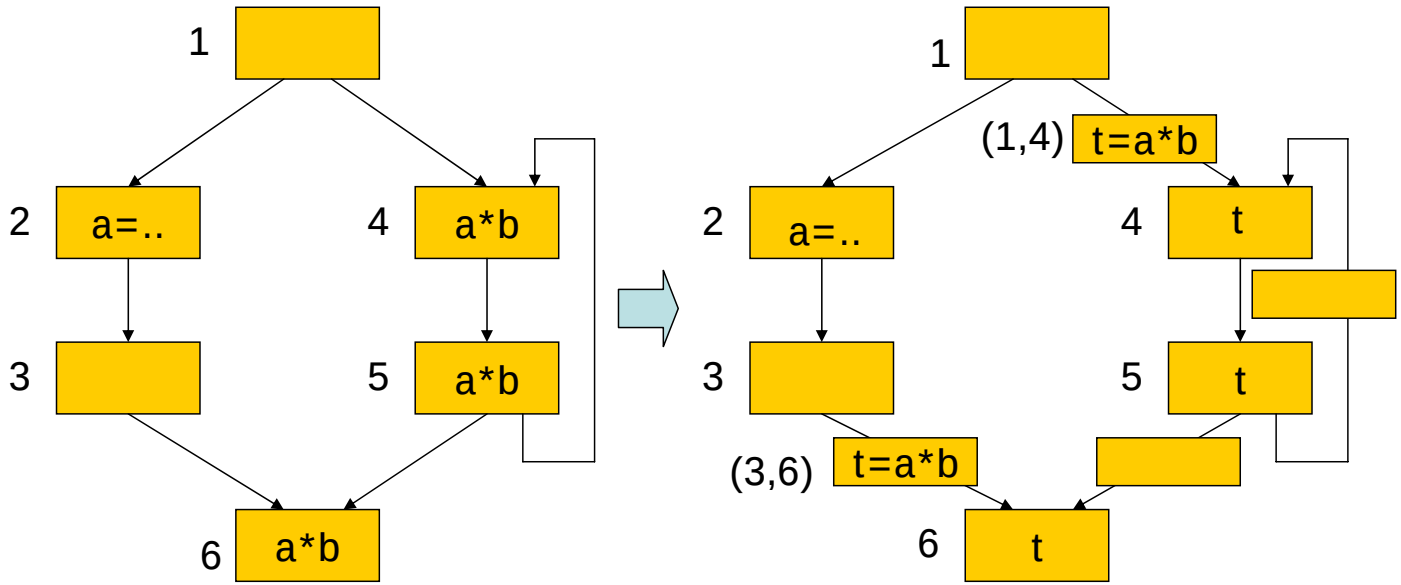
Algorithmus von Morel und Renvoise 1979



- 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**

Quelle: Dhamdhere, Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Alg. von Knoop, Ruthing und Steffen 1992



- 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)

Quelle: Dhamdhere, Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Konzepte

Verfügbarkeit

TECHNISCHE UNIVERSITÄT DARMSTADT

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*

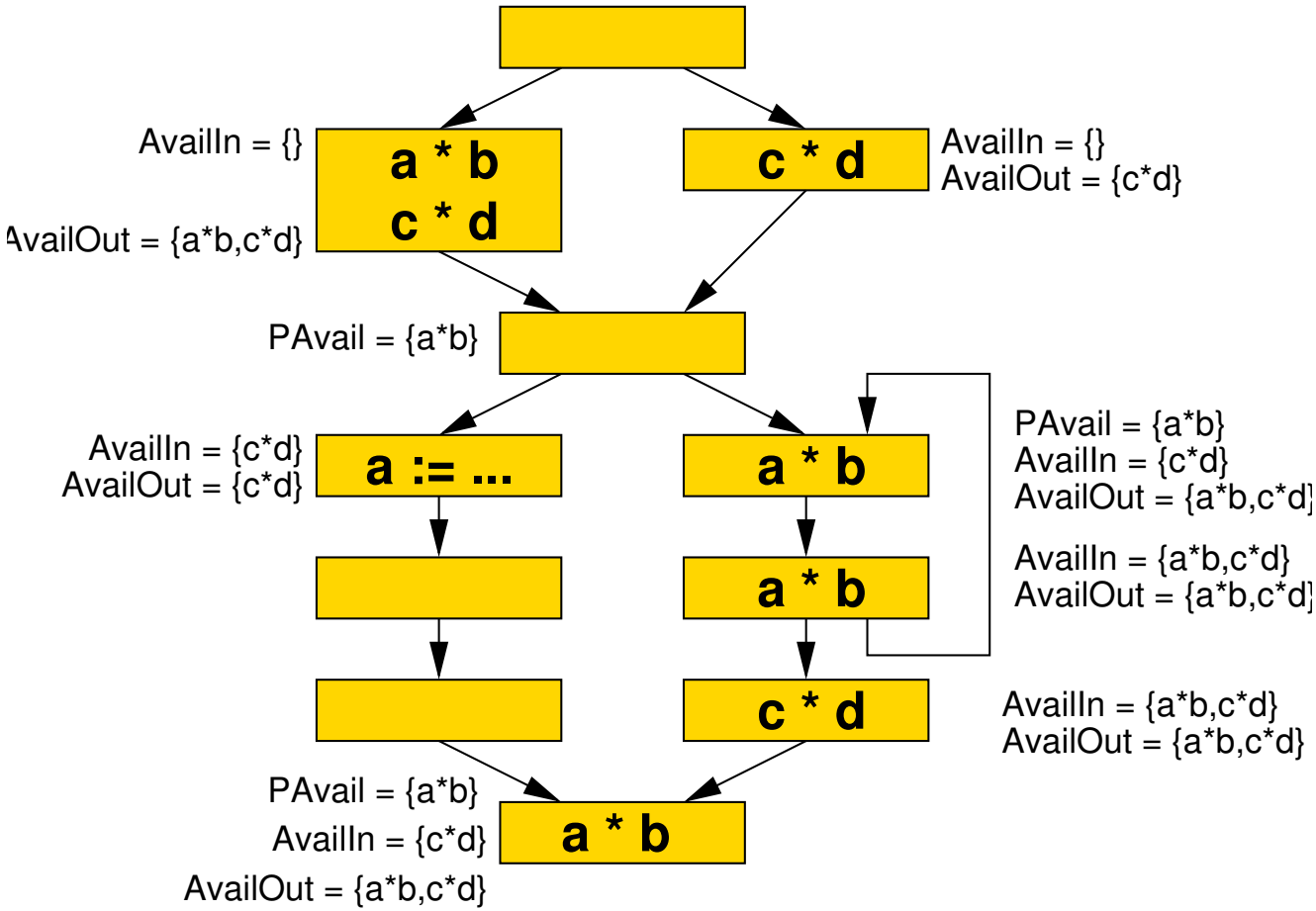
Partielle Verfügbarkeit (Partial Availability)

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

TECHNISCHE UNIVERSITÄT DARMSTADT



Lokal exponierte Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT

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

- $\rightarrow$ UEExpr
- Auch genannt lokal vorziehbar (locally anticipatable, ANTloc)

Lokal exponierte Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT

Nach unten exponierte Ausdrücke (Downward Exposed)

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

- $\rightarrow$ DEExpr
- Heisst auch: Ausdrücke sind lokal verfügbar
- (Berechnung könnte an Blockende verzögert werden, falls Ergebnis nicht noch im Block gebraucht)

Beispiel: Lokal exponierte Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT

a : = 42	
w := a + b	nicht UEExpr/ANTloc, DEExpr
x := c + d	UEExpr/ANTloc, DEExpr
y := e + f	UEExpr/ANTloc, nicht DEExpr
z := a + e	nicht UEExpr/ANTloc, nicht DEExpr
e := 23	

Globale Vorziehbarkeit

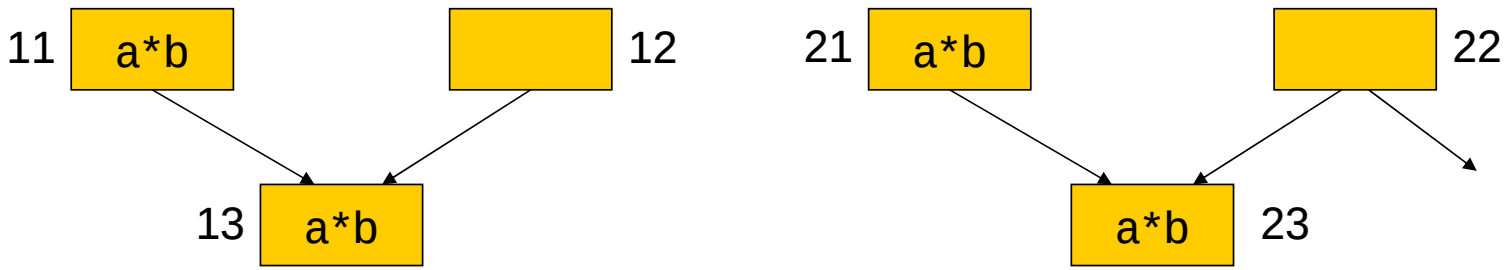
TECHNISCHE UNIVERSITÄT DARMSTADT

Vorziehbarkeit (Anticipatability)

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

➥Auch genannt: Very Busy Expression

Beispiel: Vorziehbarkeit



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

Quelle: Dhamdhere, Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Sicherheit von Berechnungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sichere Berechnung

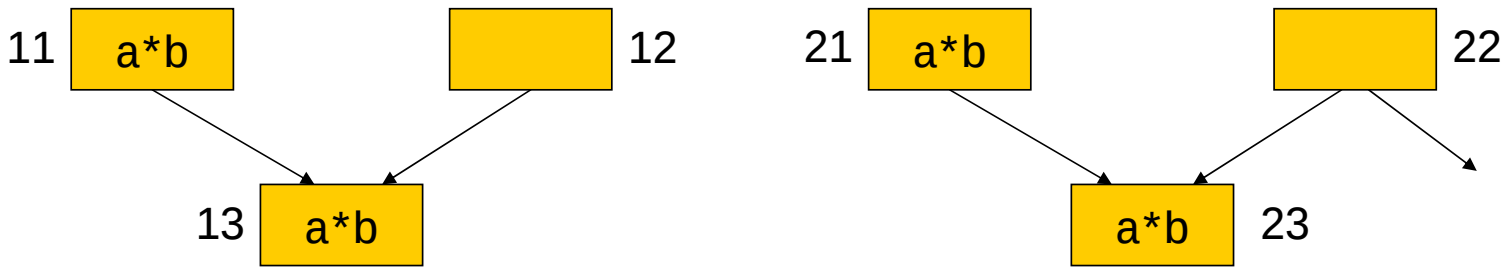
Ein Ausdruck *e* kann an der Stelle *p* sicher berechnet werden, wenn er dort bereits verfügbar ist oder dorthin vorgezogen werden kann.

➥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

TECHNISCHE UNIVERSITÄT DARMSTADT

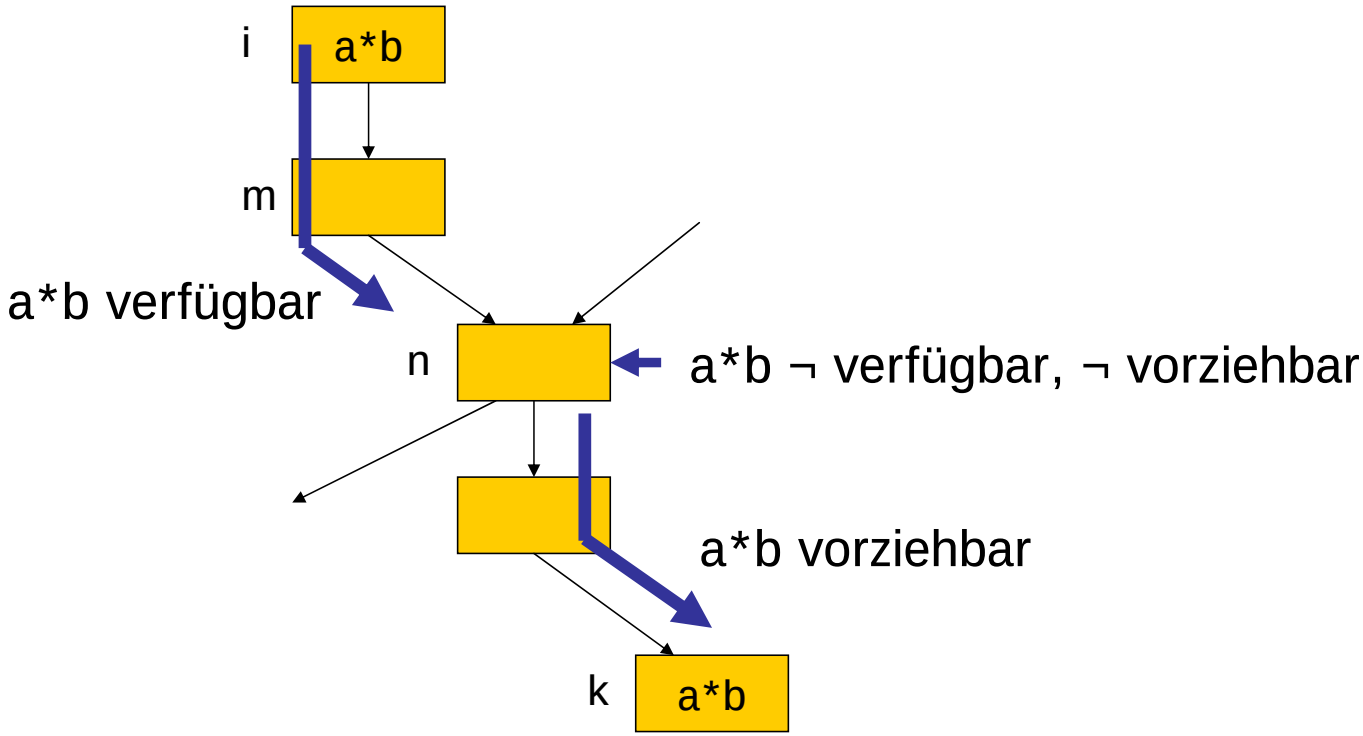


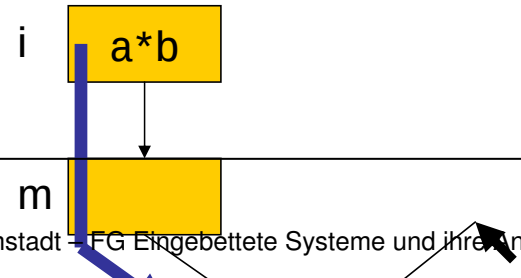
- Neuberechnung von **a*b** in Block 12 ist sicher
- **a*b** in Block 22 ist unsicher
- **a*b** in Kante (Block 22, Block 23) wäre sicher

Quelle: Dhamdhere ‚Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Beispiel: Verfügbar, vorziehbar, sicher

TECHNISCHE UNIVERSITÄT DARMSTADT





Transparenz und Leere

TECHNISCHE UNIVERSITÄT DARMSTADT

Transparenz

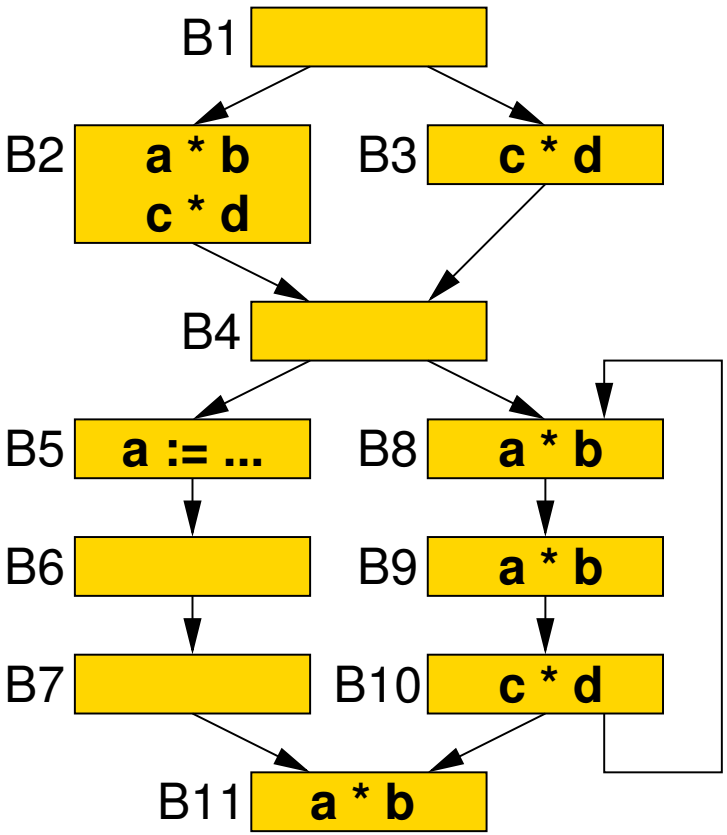
Ein Block **B** ist transparent in Hinblick auf einen Ausdruck *e*, wenn er selbst keine Zuweisungen an Operanden von *e* enthält.

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: empty(**B**) = TRUE bezüglich *e*

Beispiel: Transparenz



- Für **a*b**
  - Alle Blöcke ausser **B5** sind transparent
  - $b \in \{$ **B1, B3, B4, B6, B7, B10** $\}$: empty(b) = TRUE
- Für **c*d**
  - Alle Blöcke sind transparent
  - $b \in \{$ **B1, B4, B5, B6, B7, B8, B9, B11** $\}$: empty(b) = TRUE

Anforderungen an guten PRE-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

PRE mit Eliminierungspfaden

TECHNISCHE UNIVERSITÄT DARMSTADT

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 Web-Seite
- Verfeinert zusammen mit Dheeraj Kumar 2006

Eliminierbarkeitspfad (E-Pfad)

TECHNISCHE UNIVERSITÄT DARMSTADT

Eliminierbarkeitspfad (E-Pfad, eliminatability path)

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

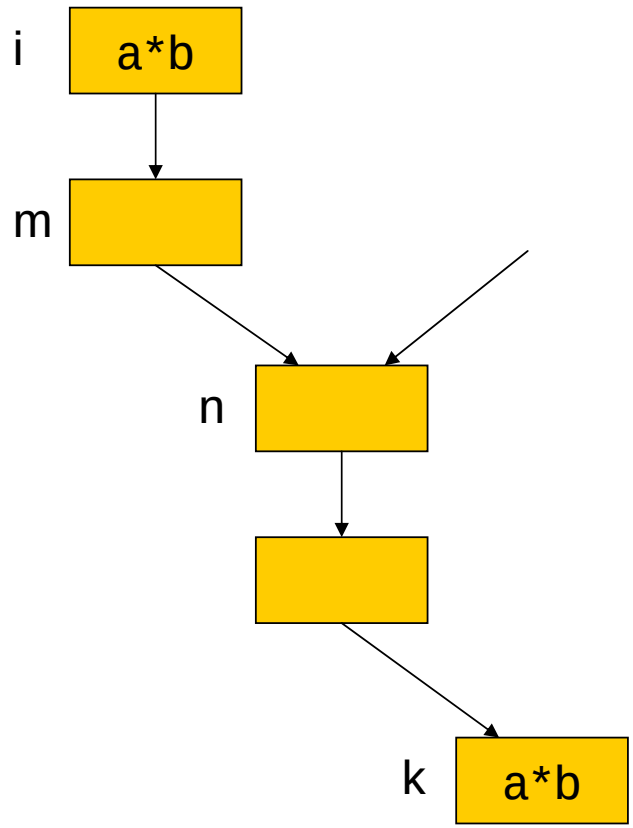
1. e lokal verfügbar in b_i und lokal vorziehbar in b_k ist
2. Für $b \in (b_i, \ldots, b_k)$ gilt: *empty(b)* = TRUE
3. e ist sicher auf jeder Ausgangskante eines Blocks $b \in [b_i, \ldots, b_k)$

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

Beispiel: E-Pfad

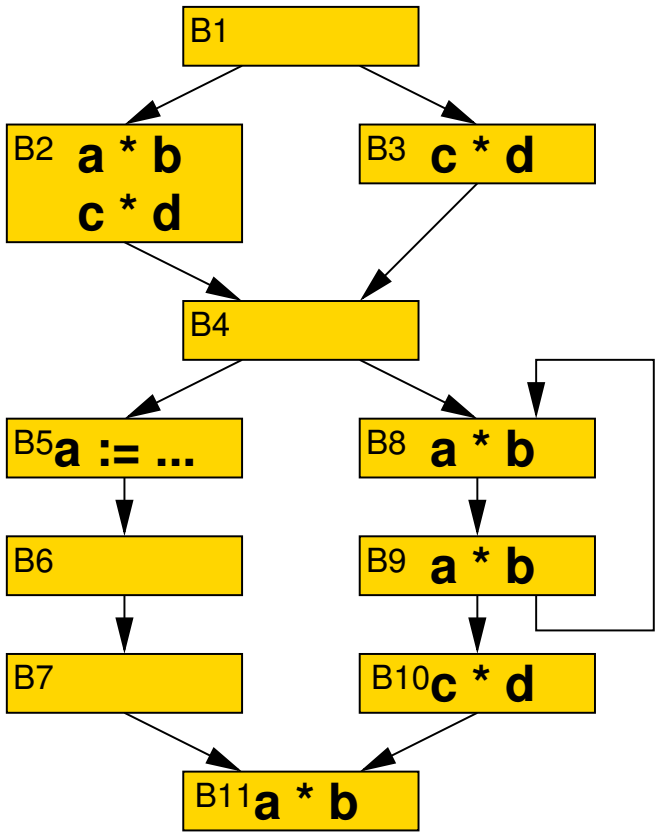
TECHNISCHE UNIVERSITÄT DARMSTADT

- **a*b** ist verfügbar an Ende von [*i* ... *m*]
- **a*b** ist vorziehbar am Ende von [*n* ... *k*)
- Berechnung von **a*b** in Block *k* ist eliminierbar



Quelle: Dhamdhere Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Beispiele für E-Pfade



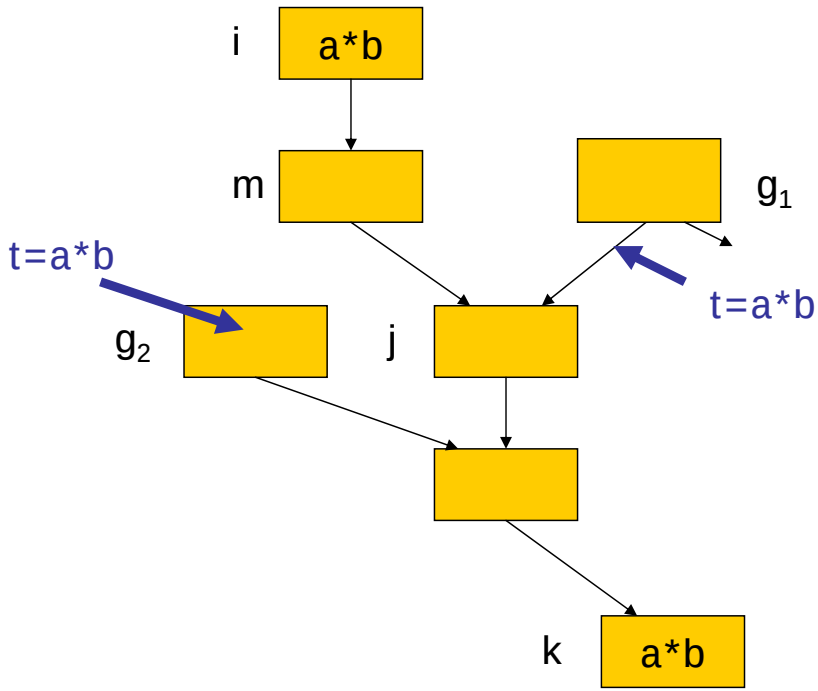
- Für **a*b**
  - $[b_8, b_9]$
  - $[b_9, b_8]$
  - $[b_9, b_{10}, b_{11}]$
  - Nicht:
    $[b_2, b_4, b_8]$ (nicht sicher),
    $[b_8, b_9, b_{10}, b_{11}]$ (nicht empty)
- Für **c*d**
  - $[b_2, b_4, b_8, b_9, b_{10}]$
  - $[b_3, b_4, b_8, b_9, b_{10}]$

Idee für die Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die ersten Vorkommen (UEExpr, vor dem Schreiben von Operanden) von e im Endblock b_k des E-Pfades $[b_i, \ldots, b_k]$ sind eliminierbar
- Das vorherige Evaluationsergebnis des Ausdrucks e wird dazu gesichert in der Variable t_e
- Wird der Pfad $(b_i, \ldots, b_k]$ 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.

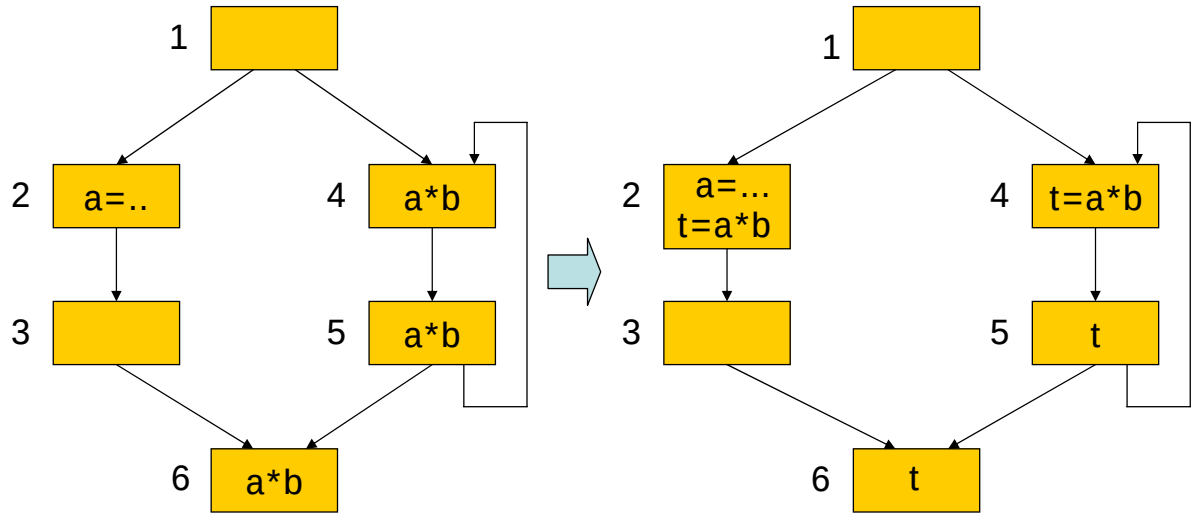
Lebenszeitoptimalität durch E-Pfade



- E-Pfad: [i, ... , k]
- Einfügen von Berechnungen von e und Kopieren
  - In Kante (g_1, j) und Block g_2

Quelle: Dhamdhere Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Vergleich mit Ergebnis nach MRA



- E-Pfade: [4, 5], [5, 4], [5, 6]; nicht: [4, 5, 6] (da nicht empty)
- 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 entfernbar sein
    $\rightarrow$ Kante (1,4) aufteilen, dort Berechnen und Kopieren

Quelle: Dhamdhere Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Aktionen während Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Wert von *e* sichern

Füge Anweisung $\texttt{t}_e$ `:=` e vor bestehender Berechnung von e ein und ersetze Berechnung durch t_e

Neue Berechnung von *e* einfügen

Füge Anweisung $\texttt{t}_e$ `:=` e ein

Eliminiere redundante Berechnung von *e*

Ersetze e durch t_e

An welchen Stellen diese Aktionen ausführen?

Neue Notation

Prädikatenschreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Wahr für alle am Ende eines Blocks *b* verfügbaren Ausdrücke *e*

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

Wahr für alle in den Block *b* vorziehbaren Ausdrücke *e*

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

Wahr für alle Ausdrücke *e*, für die beides gilt

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

Wahr für alle Ausdrücke, für die mindestens eines gilt

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

Konfluenzoperatoren in Prädikatenschreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

Konjunktion (am Beispiel von Copy Propagation)

$cpin_c(b) = \prod_p cpout_c(p)$ = TRUE für Kopie $c \Leftrightarrow c \in \bigcap_{p \in pred(\mathrm{b})} \textsc{CPout}(p)$

Disjunktion (am Beispiel von Liveness-Analyse)

$liveout_v(b) = \sum_s livein_v(s)$ = TRUE für Variable $v \Leftrightarrow v \in \bigcup_{s \in succ(\mathrm{b})} \textsc{LiveIn}(s)$

Prädikate darstellen als Bitvektor

TECHNISCHE UNIVERSITÄT DARMSTADT

Für alle Ausdrücke eindeutige Zuordnung festlegen

...

a*b $\rightarrow$ Bit 1

c*d $\rightarrow$ 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}	$\{\mathbf{c} * \mathbf{d}\}$
10	x_{a*b}, $\neg x_{c*d}$	$\{\mathbf{a} * \mathbf{b}\}$
11	x_{a*b}, x_{c*d}	$\{\mathbf{a} * \mathbf{b}, \mathbf{c} * \mathbf{d}\}$

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

Beispiel: Verknüpfung von Prädikaten

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuordnung von Ausdrücken e an Bits

a*b $\rightarrow$ Bit 0
c*d $\rightarrow$ Bit 1
a*d $\rightarrow$ Bit 2
...

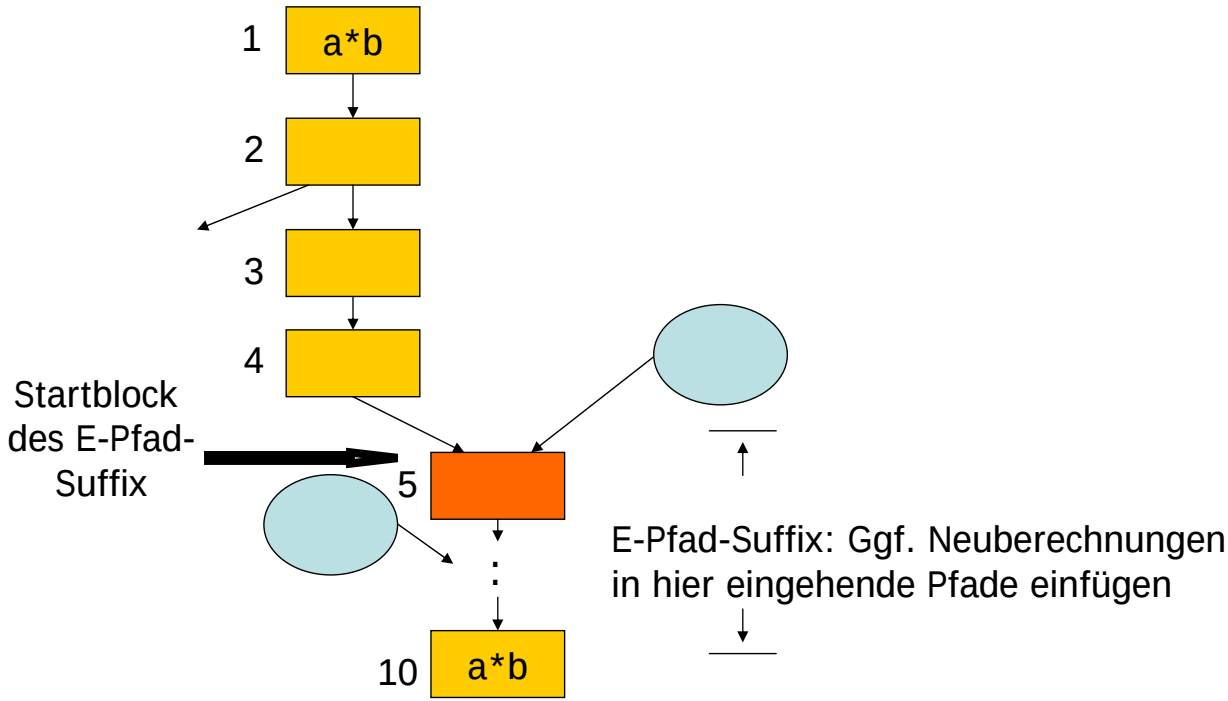
Gegeben: Zwei Prädikate x und y bezüglich Ausdruck e
Annahme: $x = 101, y = 011$

$$x \cdot y = x \& y = 001$$
$$x + y = x | y = 111$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Bestimmen von E-Pfaden

Anatomie eines E-Pfades



$b \in (1, 2]$ avail(b) $\cdot \neg$ant(b)

$b \in [3, 4]$ avail(b) $\cdot$ ant(b)

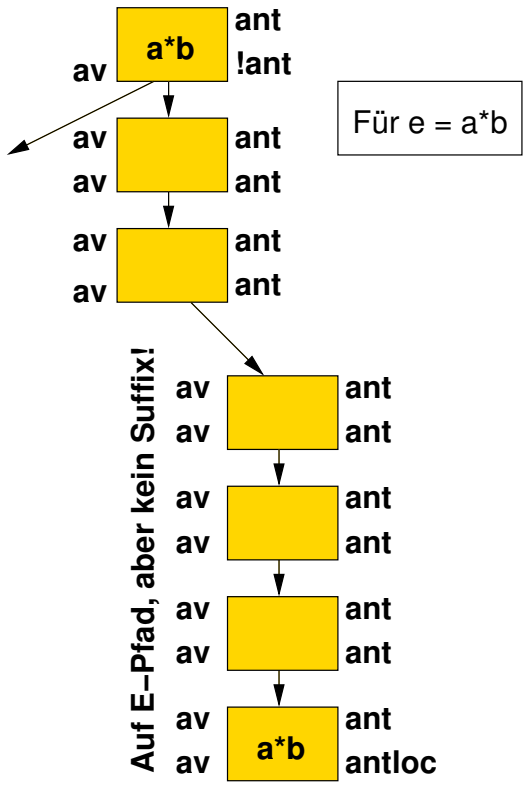
$b \in [5, \ldots, 10]$ $\neg$avail(b) $\cdot$ ant(b) E-Pfad-Suffix

Quelle: Dhamdhere, Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Beispiel: Leerer E-Pfad-Suffix

TECHNISCHE UNIVERSITÄT DARMSTADT

Anforderung an E-Pfad-Suffix: ¬avail(b) · ant(b)

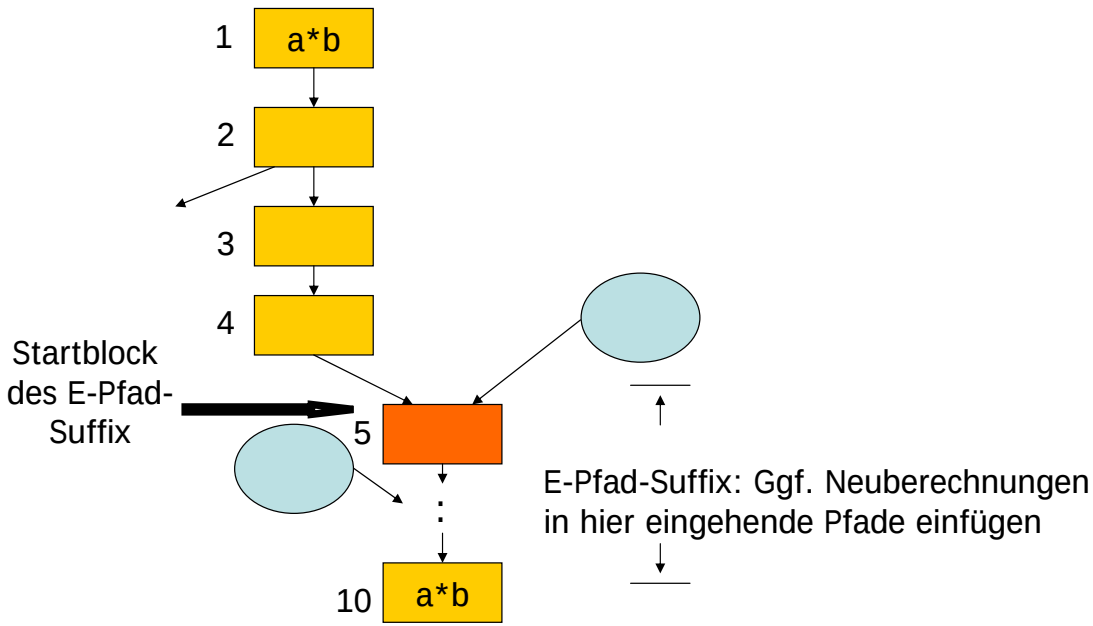


Algorithmus zum Bestimmen von E-Pfaden

TECHNISCHE UNIVERSITÄT DARMSTADT

- E-Pfad $[b_i, \ldots, b_k]$ hat drei (ggf. leere) Segmente
  - avail(b) $\cdot \neg$ant(b)
  - avail(b) $\cdot$ ant(b)
  - $\neg$avail(b) $\cdot$ 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
  avail(b) $\cdot$ ant(b)
  avail(b) $\cdot \neg$ant(b)
  bis Start des E-Pfads b_i erreicht
- Suche vorwärts von b_m durch Segment
  $\neg$avail(b) $\cdot$ ant(b)
  bis Ende des E-Pfades erreicht

Aktionen nach Bestimmen des E-Pfades



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(en) von e durch t_e

Parallele Lösung der E-Pfad-Bestimmung

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Datenflussproblem

Art des Datenflussproblems

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Vorberechenbare Eigenschaften

TECHNISCHE UNIVERSITÄT DARMSTADT

Ausdruck ist lokal verfügbar (locally available)

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

➥$e \in \text{DEEXPR(b)}$

Ausdruck ist lokal vorziehbar (locally anticipatable)

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

➥$e \in \text{UEEXPR(b)}$

Block ist transparent für Ausdruck

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

➥$e \notin \text{EXPRKILL(b)}$

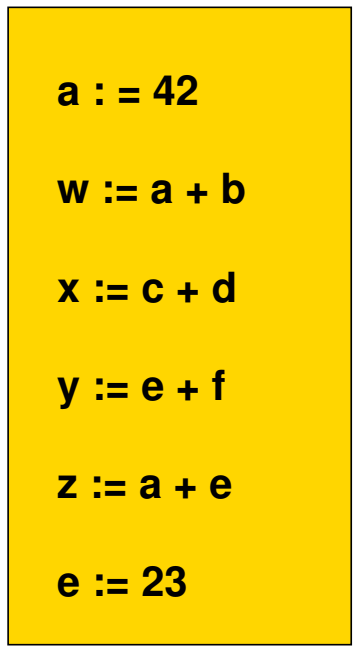
Beispiel für vorberechenbare Eigenschaften

TECHNISCHE UNIVERSITÄT DARMSTADT

Bit 5: a+b
Bit 4: c+d
Bit 3: e+f
Bit 2: a+e
Bit 1: q+r
Bit 0: x+y

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

B1



nicht UEExpr/ANTloc, DEExpr

UEExpr/ANTloc, DEExpr

UEExpr/ANTloc, nicht DEExpr

nicht UEExpr/ANTloc, nicht DEExpr

Verfügbare Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT

$\text{availin}_e(b)$ e ist verfügbar bei Eintritt in b

$\text{availout}_e(b)$ e ist verfügbar bei Austritt aus b

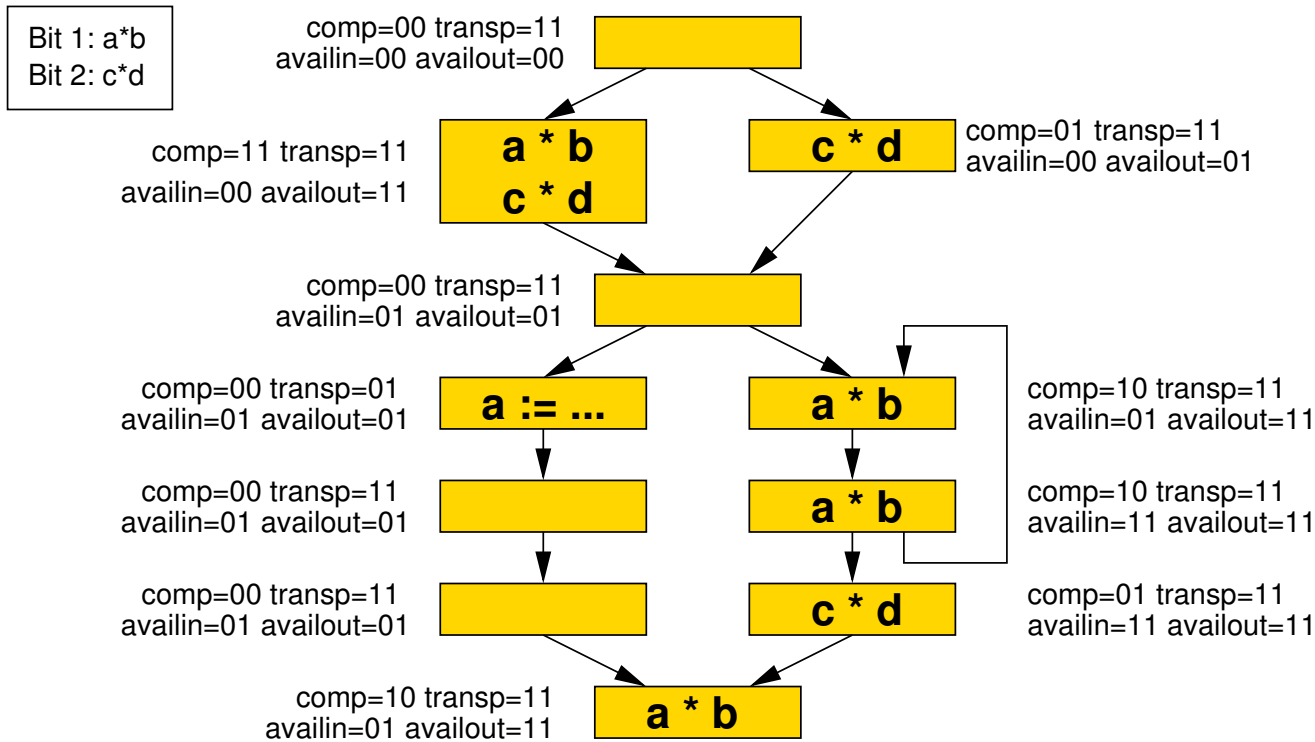
Berechnung

$$\text{availin}_e(b) = \prod_p \text{availout}_e(p)$$

$$\text{availout}_e(b) = \text{availin}_e(b) \cdot \text{transp}_e(b) + \text{comp}_e(b)$$

Beispiel verfügbare Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT



$$\text{availin}_e(b) = \prod_p \text{availout}_e(p)$$

$$\text{availout}_e(b) = \text{availin}_e(b) \cdot \text{transp}_e(b) + \text{comp}_e(b)$$

Vorziehbare Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

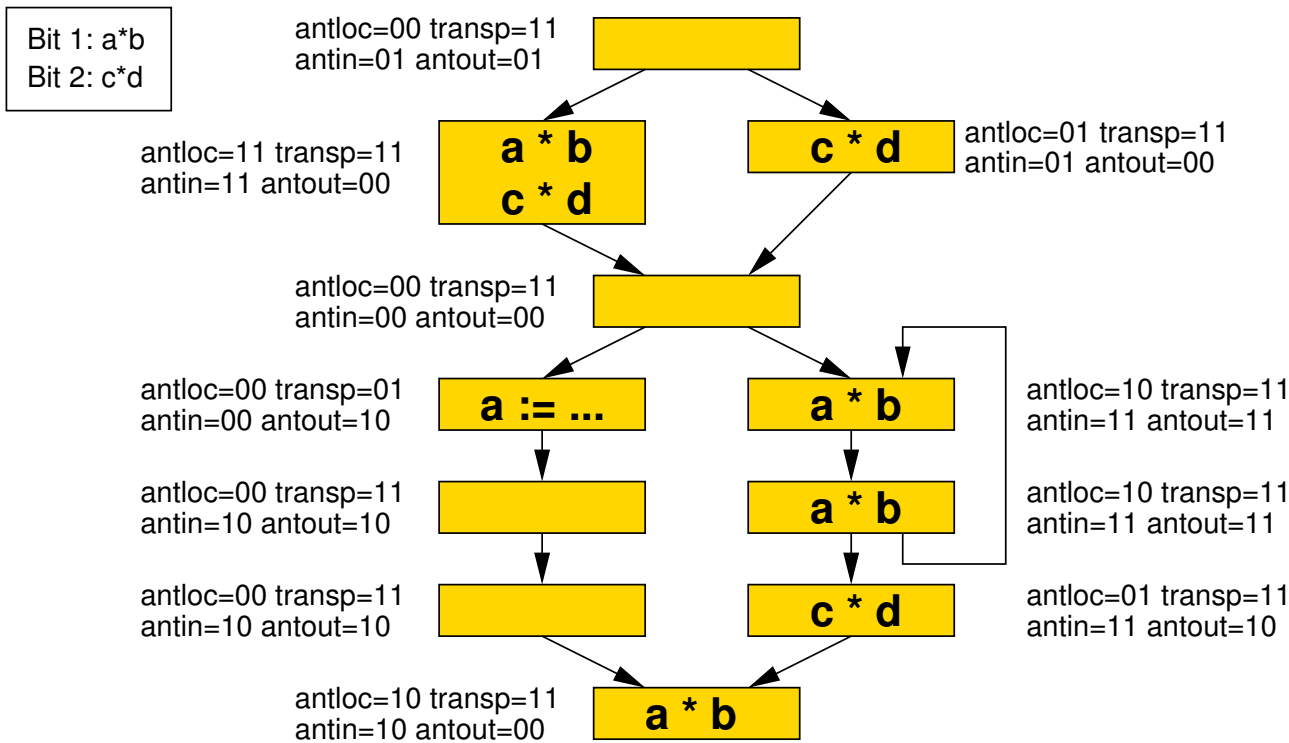
Berechnung

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

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

Beispiel vorziehbare Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT



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

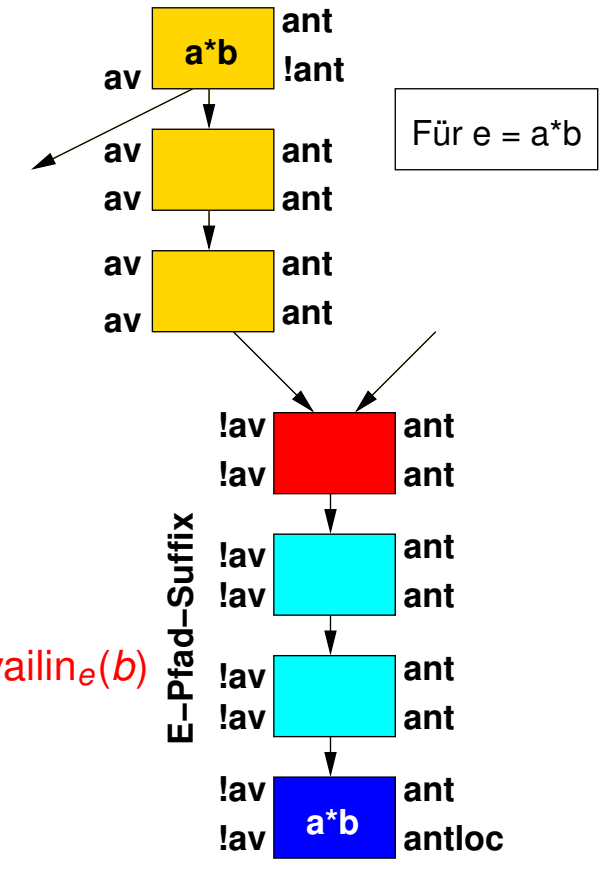
$$\text{antout}_e(b) = \prod_s \text{antin}_e(s)$$

Bestimmen von Blöcken in E-Pfad-Suffix

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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



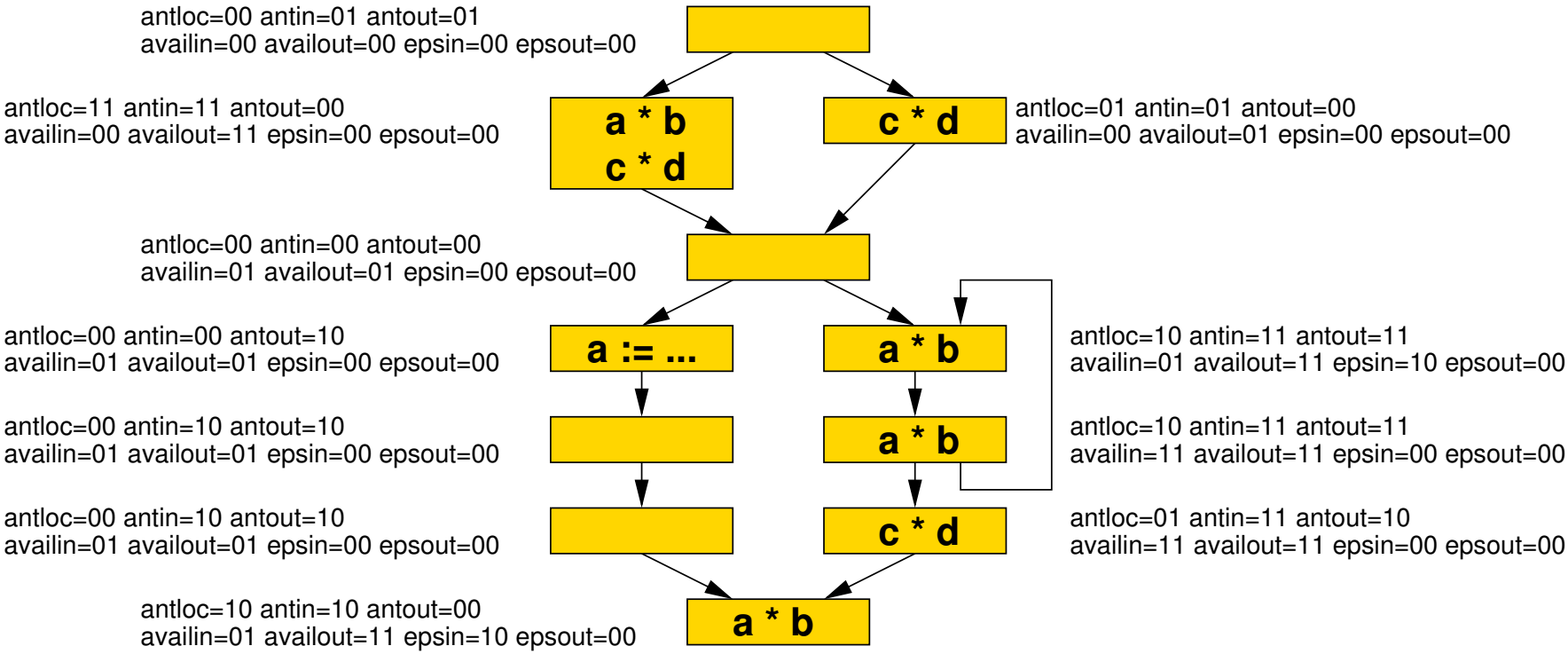
Berechnung

$$\text{epsin}_e(b) = \left(\sum_p \left(\text{availout}_e(p) + \text{epsout}_e(p)\right)\right) \cdot \text{antin}_e(b) \cdot \neg\text{availin}_e(b)$$

$$\text{epsout}_e(b) = \text{epsin}_e(b) \cdot \neg\text{antloc}_e(b)$$

Beispiel E-Pfad-Suffix

TECHNISCHE UNIVERSITÄT DARMSTADT



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

Idee: Bestimmen der Blöcke der redundanten Berechnungen

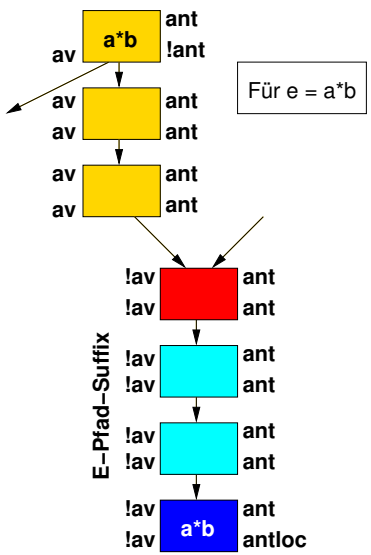
TECHNISCHE UNIVERSITÄT DARMSTADT

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

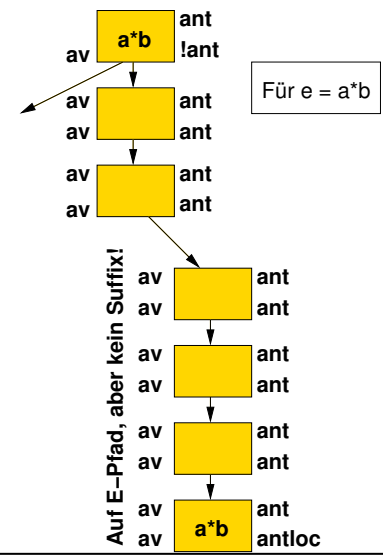
$\Leftrightarrow \text{epsin}_e(b) \cdot \text{antloc}_e(b)$



E-Pfad-Suffix $= \emptyset$

Endblock des E-Pfades

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



Erkennen von redundanten Berechnungen

TECHNISCHE UNIVERSITÄT DARMSTADT

$\text{redund}_e(b)$ Berechnung von e in b ist redundant und kann durch $\texttt{t}_e$ ersetzt werden

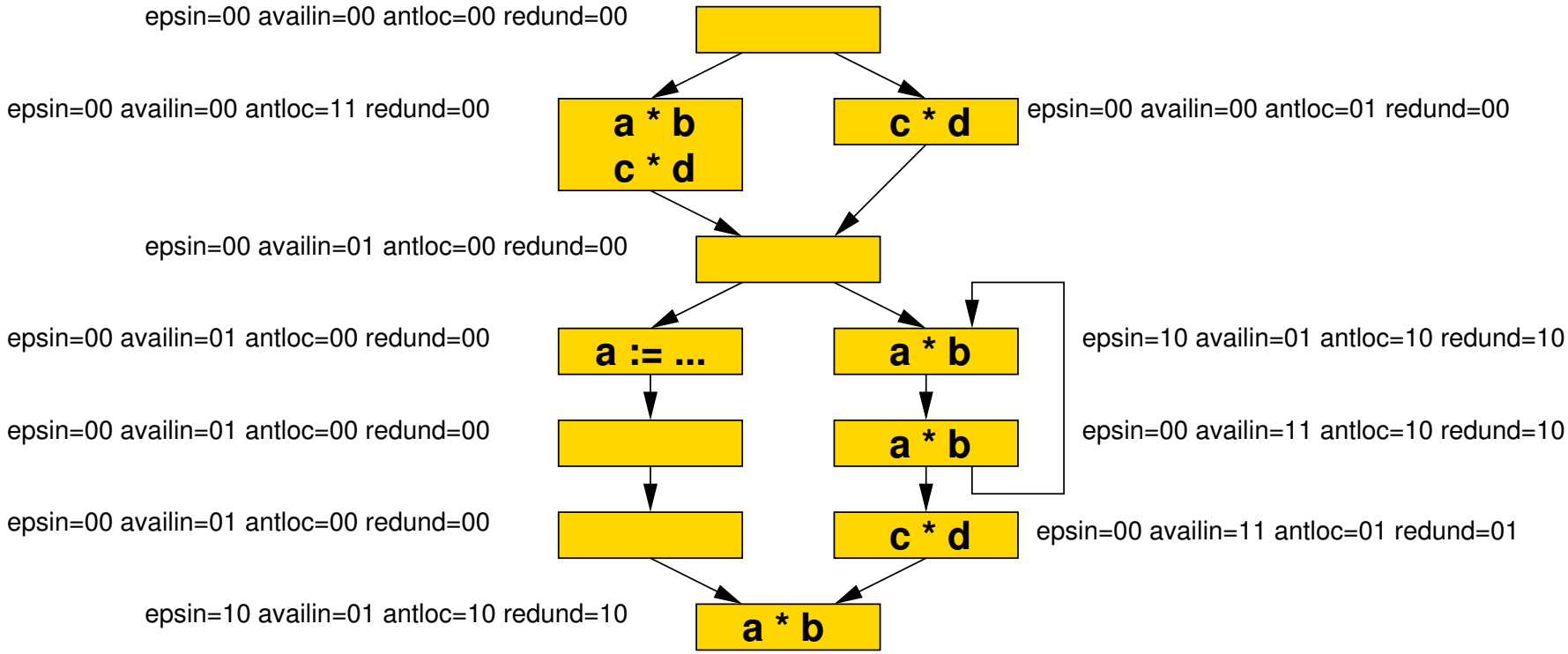
Berechnung

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

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

Beispiel Redundanz

TECHNISCHE UNIVERSITÄT DARMSTADT

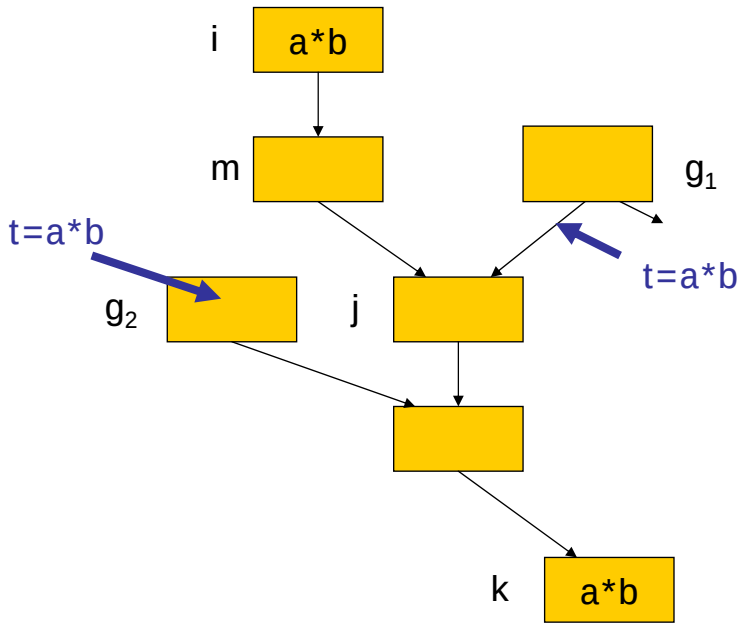


$$\text{redund}_e(b) = (\text{epsin}_e(b) + \text{availin}_e(b)) \cdot \text{antloc}_e(b)$$

Einfügestellen für neue Berechnungen

TECHNISCHE UNIVERSITÄT DARMSTADT

An Eintrittspunkten in den Pfad!



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

Quelle: Dhamdhere, Code optimization by partial redundancy elimination using Eliminatability paths (E-paths)

Einfügestellen für neue Berechnungen bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

$b_h \notin$ E-Pfad, $b_j \in$ E-Pfad $(b_i, \ldots, b_k]$

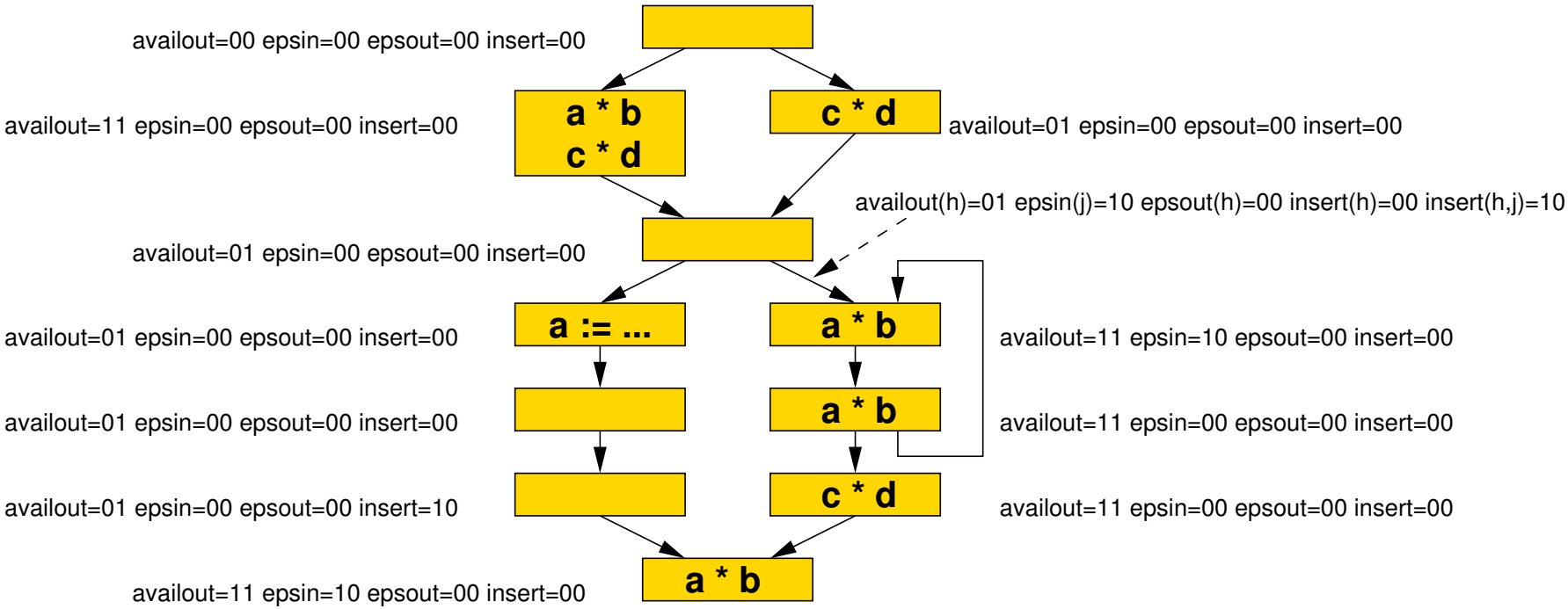
Berechnung

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

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

Beispiel Einfügen von Berechnungen

TECHNISCHE UNIVERSITÄT DARMSTADT



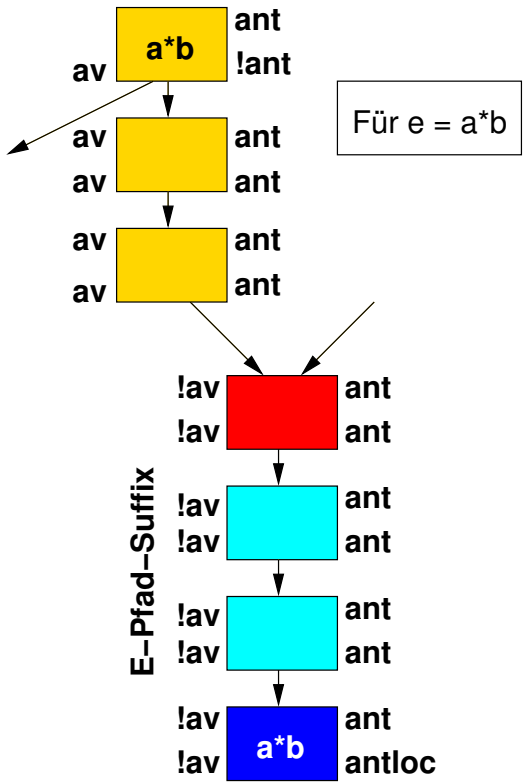
$$\text{insert}_e(b_h) = \neg\text{availout}_e(b_h) \cdot \neg\,\text{epsout}_e(b_h) \cdot \prod_s \text{epsin}_e(s)$$

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

Sichern von Berechnungsergebnissen

TECHNISCHE UNIVERSITÄT DARMSTADT

Am Anfang des E-Pfades!

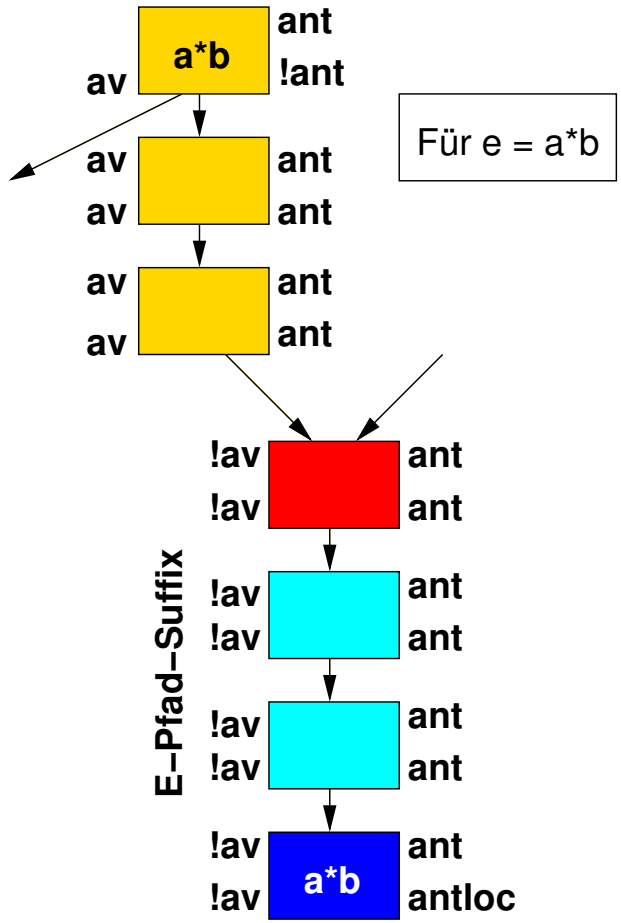


➥Startblock des E-Pfades bestimmen!

Idee zur Bestimmung des Startblocks

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



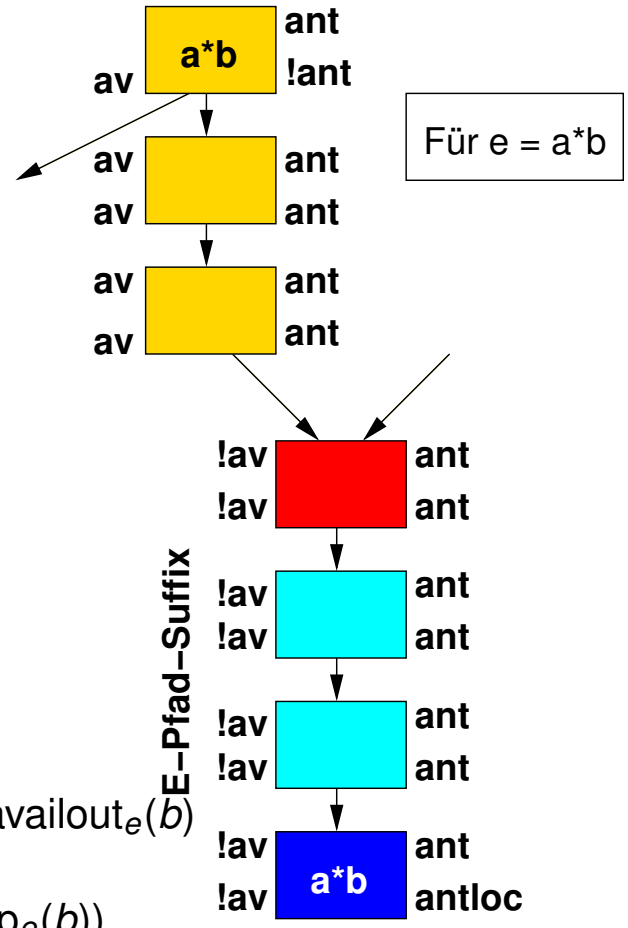
Berechnung des Startblocks

TECHNISCHE UNIVERSITÄT DARMSTADT

$\text{svupin}_e(b)$ Sichere Ergebnis für e in Block vom Ausgang zum Eingang von b

$\text{svupout}_e(b)$ Sichere Ergebnis für e über Blockgrenze zum Ausgang von b

$\text{save}_e(b)$ Sichere Ergebnis in $\texttt{t}_e$ in Block b



Berechnung

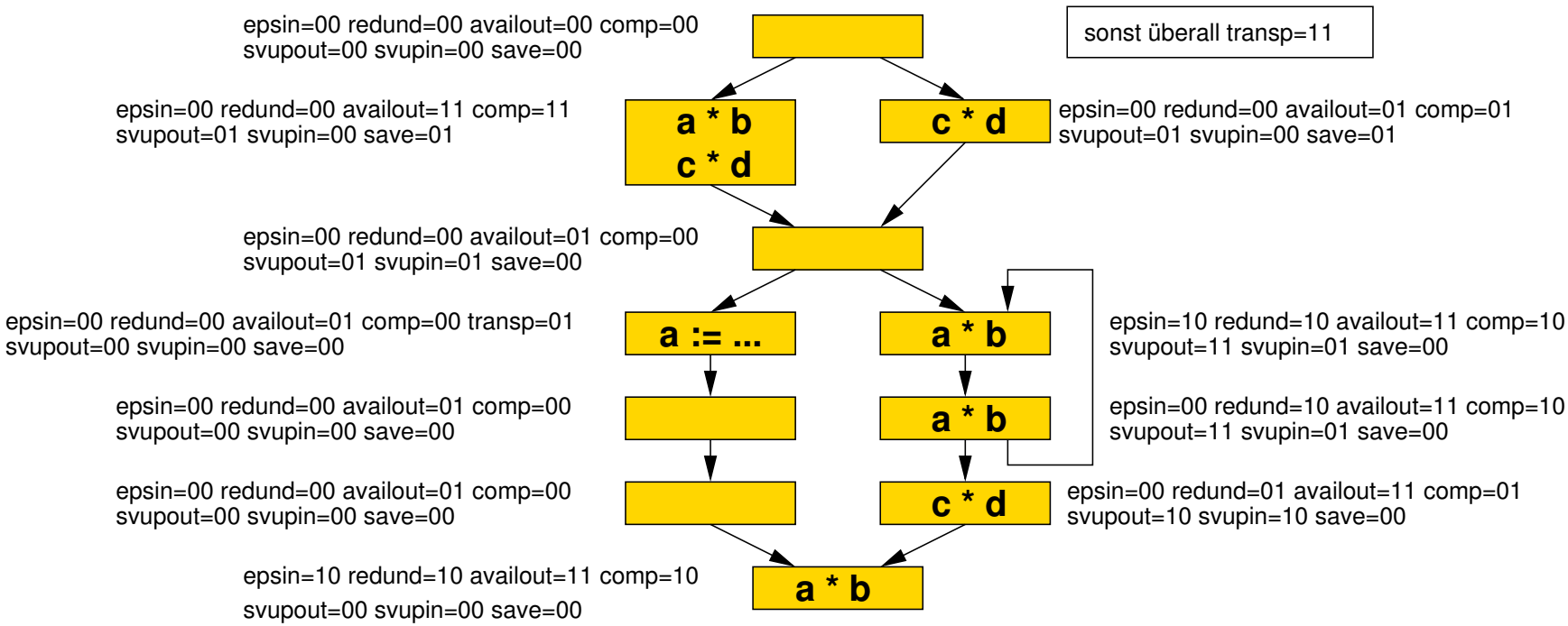
$$\text{svupout}_e(b) = (\sum_s (\text{epsin}_e(s) + \text{redund}_e(s) + \text{svupin}_e(s))) \cdot \text{availout}_e(b)$$

$$\text{svupin}_e(b) = \text{svupout}_e(b) \cdot\neg\ \text{comp}_e(b)$$

$$\text{save}_e(b) = \text{svupout}_e(b) \cdot \text{comp}_e(b) \cdot\neg\ (\text{redund}_e(b) \cdot \text{transp}_e(b))$$

Beispiel Bestimmung der E-Pfad-Startblöcke

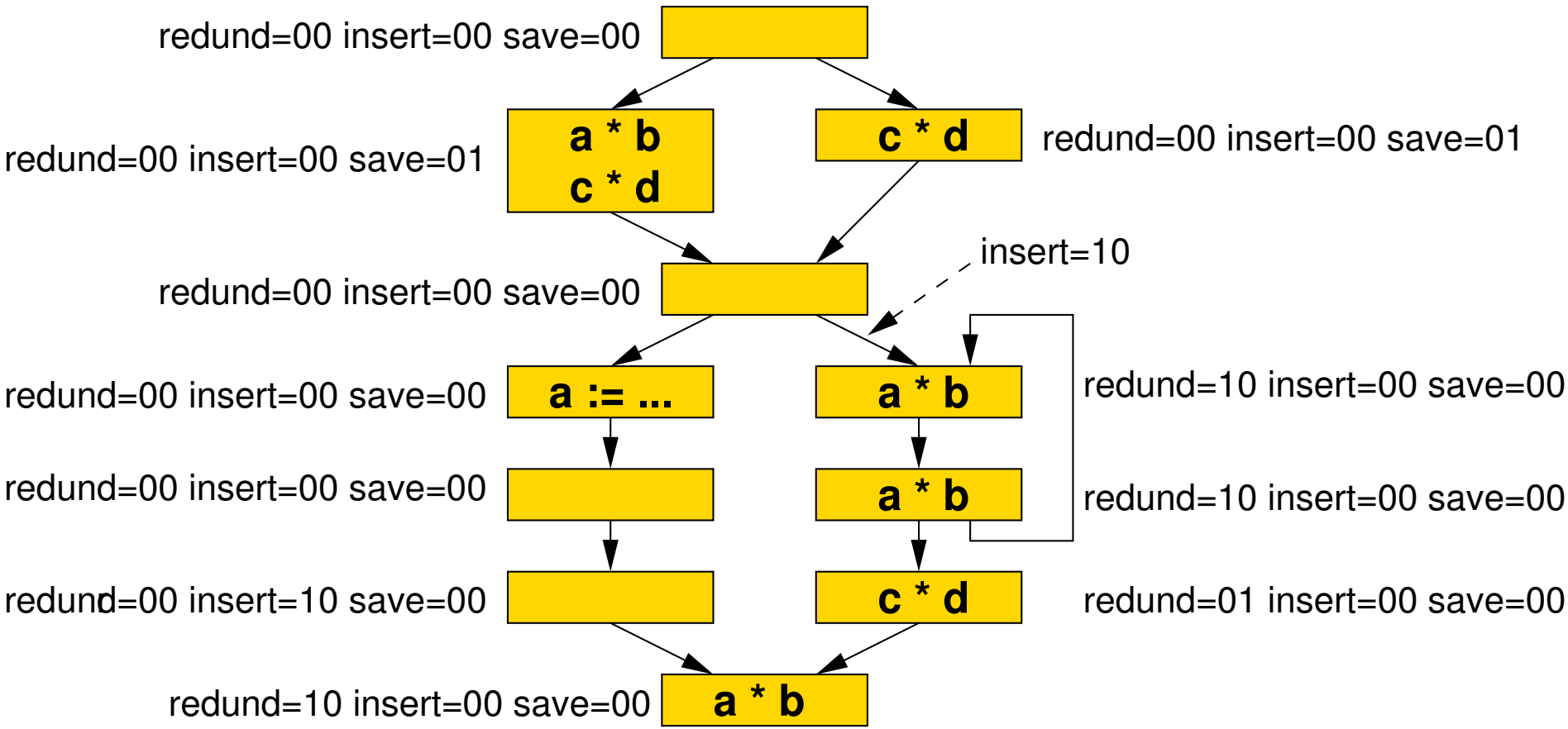
TECHNISCHE UNIVERSITÄT DARMSTADT



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

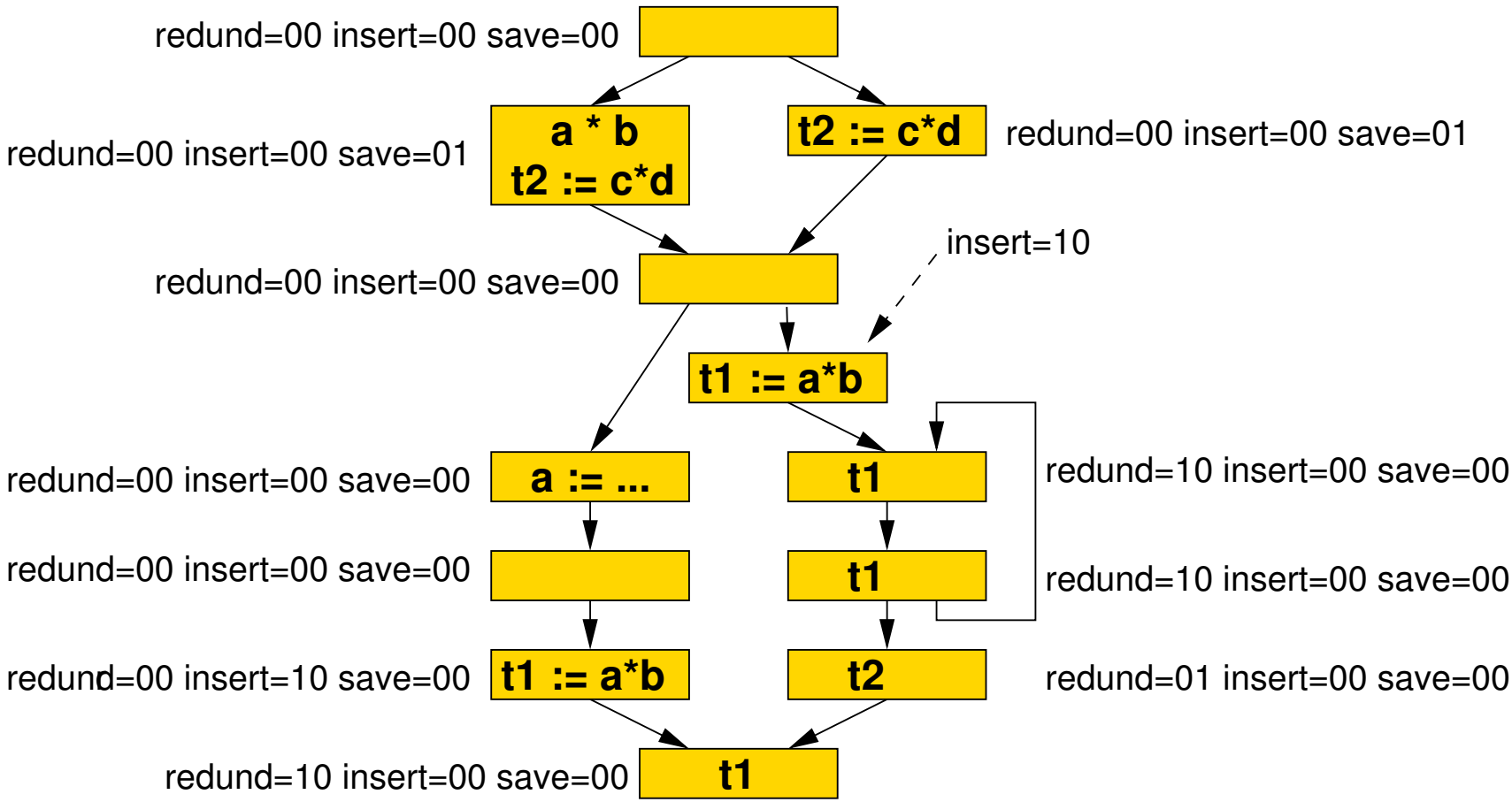
Beispiel Zusammenfassung Datenflussanalyse

TECHNISCHE UNIVERSITÄT DARMSTADT



Beispiel PRE-Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT



Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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 $\rightarrow$ SSA-PRE
  - Verfeinerte Stellung des Datenflussproblems
    $\rightarrow$ D. Kumar 2006, geringerer Rechenaufwand