

Optimierende Compiler

Einführung in Code-Optimierung

Andreas Koch

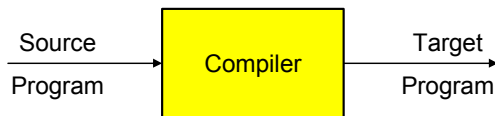
FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2011

Organisatorisches

- Ab jetzt vorgehen nach Cooper & Torczon
 - Daraus ab jetzt auch die meisten Zeichnungen
- Geplant: Behandle Kapitel 8-10
- Unterstützt durch Papers
 - Werden als PDF auf Web-Seite gelegt

Einleitung



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

➔ Ziel: Effizienz eines erfahrenen Assembler-Programmiers

Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um

Sicherheit Die Bedeutung des Programmes (hier: extern beobachtetes Verhalten) darf **nicht** verändert werden.

Profitabilität Die Optimierung muß in Bezug auf das angestrebte Ziel eine **ausreichende** Verbesserung bringen.

Risiko Welche **negativen** Effekte kann die Optimierung nach sich ziehen?

Anwendbarkeit Gibt es **ausreichend** viele Stellen, an denen die Optimierung angebracht werden kann?

Redundante Ausdrücke

Ursprünglicher Code

```
m := 2 * y * z;  
n := 3 * y * z;  
o := 2 * y - z;
```

Umgeschriebener Code

```
t0 := 2 * y;  
m := t0 * z;  
n := 3 * y * z;  
o := t0 - z;
```

A. Koch

Redundanter Ausdruck

Ein Ausdruck $x \text{ op } y$ ist an einer Position L redundant, wenn er in jedem Fall vor L berechnet wurde, und zwischen der Berechnung und L die Operanden x und y nicht verändert wurden.

Basisblock (BB)

Längste Folge von Anweisungen ohne Kontrollfluß.

Beispiel:

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

Basisblöcke:

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

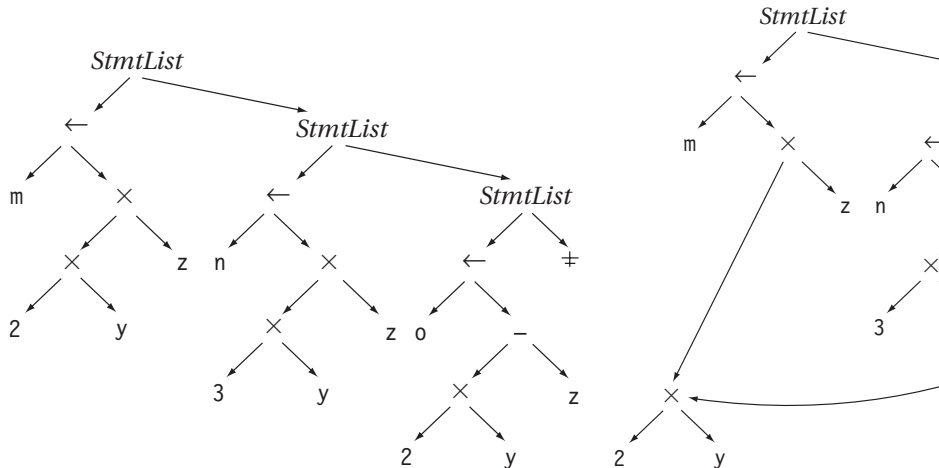
```
c := a - 46;  
d := b * 15;
```

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

Erster Ansatz: AST-Ebene

Erkennen gleicher Teilbäume, dann Erweiterung auf DAG

A. Koch



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

A. Koch

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2*y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

Braucht Mechanismus, um Zuweisungen zu beachten!
Idee

A. Koch

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung (\rightarrow SSA)
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei m_0 nun als bekannt den Ausdruck $2*y_0$ ein
- Unterbindet Wiederverwendung als $2*y_1$

CFG-basierte Redundanzeliminierung mit Wertnumerierung

Andere Zwischendarstellung: Kontrollflußgraphen

- Relationen zwischen Basisblöcken
 - Bedingte Ausführung, Schleifen
- Verwaltet nun Folgen von Anweisungen in Basisblöcken
 - Häufige IR, z.B. in SUIF, Scale, COINS, GCC (ab 4.x)
- Reihenfolge leichter umzustellen

Nun keine Analyse des AST mehr!

Idee

- Verwalte nun nicht Variablen**namen** in Ausdrücken
- ... sondern direkt ihre **Werte**
 - Jeder zur Laufzeit berechnete Wert wird durchnummeriert
 - Der Wert zweier Ausdrücke e_1 und e_2 bekommt dieselbe Nummer
 - ... wenn e_1 und e_2 **beweisbar immer** denselben Wert liefern

- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

A. Koch

Implementierung mit Hash-Table

- Hashe Textdarstellung von Variablen und Konstanten (**.spelling**)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und **Wertnummern**
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.

Wie Kollisionen handhaben?

A. Koch

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke

Beispiel Value Numbering

```
x := a * d;  
y := a;  
z := y * d;
```

A. Koch

Ausdruck	... auf Wertnummern	Wertnummer Ergebnis
a	-	1
d	-	2
a * d	1*2	3
x	3	3
a	1	1
y	1	1
y * d	1*2	3
z	3	3

Wichtig: Zahlen sind **keine** Werte, sondern Wertnummern!

Für jeden Ausdruck e der Form $r_e \leftarrow x_e \text{ op}_e y_e$ im Block

- 1 Bestimme Wertnummern von x_e, y_e
- 2 Bestimme Hash-Wert für x_e, y_e, op_e
- 3 Hash-Wert bekannt?
 - Ja: ersetze e durch Kopie von bekanntem Ausdruck, trage dessen Wertnummer für r_e ein.
 - Nein: Trage Hash-Wert neu in Tabelle ein, vergebe neue Wertnummer und trage diese für r_e ein.

- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für “konstanter Wert”
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

Algebraische Eigenschaften

$x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x\div 1$, $x-x$, $x\neq 0$,
 $x\div x$, $x\vee 0$, $x \wedge 0xFF\dots FF$,
 $\max(x, \text{MAXINT})$, $\min(x, \text{MININT})$,
 $\max(x, x)$, $\min(y, y)$, and so on ...

- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen

Bisher: Erkennen von Redundanzen, jetzt auch
Umschreiben des Codes

Eingabe-Code

```
a := b + c;  
b := a - d;  
c := b + c;  
d := a - d;
```

Value Numbering

```
a3 := b1 + c2;  
b5 := a3 - d4;  
c6 := b5 + c2;  
d5 := a3 - d4;
```

Umschreiben

```
a := b + c;  
b := a - d;  
c := b + c;  
d := b;
```

Wertinstanzen der SSA-Form

Eingabe-Code

$a_0 \leftarrow x_0 + y_0$
* $b_0 \leftarrow x_0 + y_0$
 $a_1 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$

Value Numbering

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow x_0^1 + y_0^2$
 $a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow x_0^1 + y_0^2$

Umgeschrieben

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow a_0^3$
 $a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow a_0^3$

- Wert 3 verfügbar als a_0^3

- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

A. Koch

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

- Wenn Wiederbenutzung billiger ist als Neuberechnung
 - Übliche Annahme
 - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
 - Neuberechnung braucht immer 1+ zusätzliche Register
 - Load Immediate braucht genau 1 zusätzliches Register
 - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
 - Entfernte Operationen sind immer nützlich ($x + 0$)
 - Vereinfachung hängt von Zielmaschine ab ($2*x, x+x$)
 - Kann aber leicht berücksichtigt werden

Wo und wie ist VN anwendbar?

- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Abarbeiten der Anweisungen in Programmreihenfolge
 - *program order*
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

A. Koch

Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

Lokale Methoden

- Arbeiten auf Basisblöcken (BB)
- Für jeden Basisblock gilt
 - Alle Anweisungen werden sequentiell abgearbeitet
 - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich

Local Value Numbering (LVN)

- Jeweils ein Basisblock betrachtet
- Gute lokale Resultate
- Aber keine Arbeit über Blockgrenzen

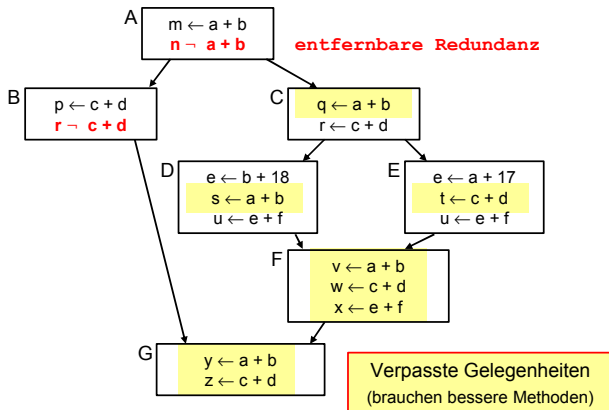
A. Koch

Erweitern des Redundanzbegriffes

Redundanz über Blockgrenzen hinweg

Ein Ausdruck $x \text{ op } y$ ist an einer Stelle L genau dann redundant, wenn er auf **jedem** Pfad vom Startknoten des CFGs zur Stelle L evaluiert worden ist und die Werte seiner Teilausdrücke x und y nicht verändert wurden.

Beispiel LVN



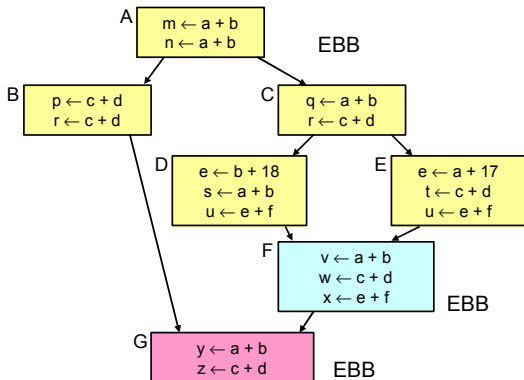
Superlokale Methoden

Superlokale Methoden 1

Arbeiten auf Extended Basic Blocks (EBBs)

- EBB $B = \{b_1, b_2, \dots, b_n\}$, mit BBs b_i
- Nur b_1 darf im CFG mehrere oder keine Vorgänger haben
- Alle anderen b_i haben genau einen Vorgänger
- EBB ist Baum aus BBs mit b_1 als Wurzel

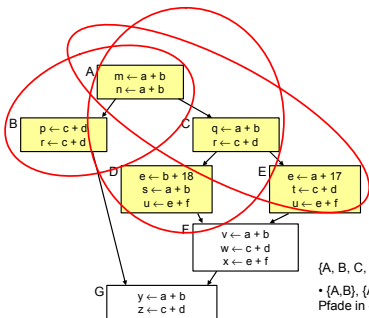
A. Koch



Superlokale Methoden 2

- Idee: Benutze **Pfade** beginnend bei b_1 durch EBB wie einen BB
- Auf jedem Pfad: Genau ein Vorgänger, baue auf dessen Analysen auf

A. Koch

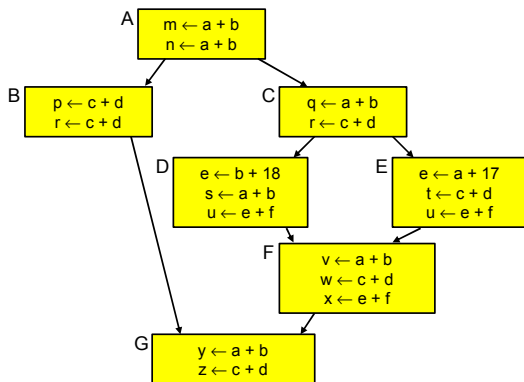


{A, B, C, D, E} ist ein EBB

• {A,B}, {A,C,D}, und {A,C,E} sind Pfade in {A, B, C, D, E}

{F} und {G} sind auch EBBs

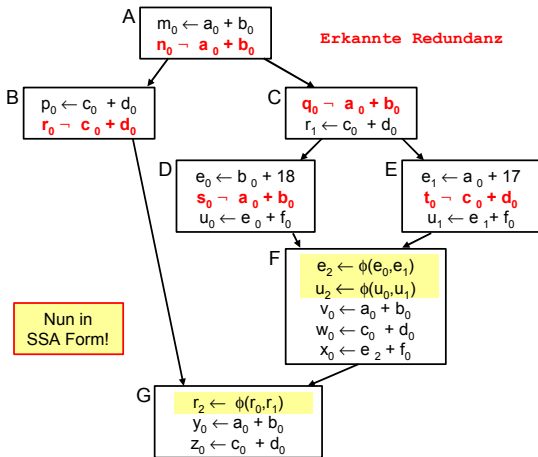
• ... haben aber nur triviale Pfade



- Idee: Wende **lokale** Methode auf jeden **Pfad** an
- Hier: Bearbeite (A, B) , (A, C, D) , (A, C, E)
- Baue auf Ergebnisse des Vorgängers auf
- **Aber**: Hilft noch nicht für F und G (woher kommt man?)

- Vermeide mehrfache Neuberechnung gleicher Daten
 - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
 - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:
Symbol-Tabelle mit Geltungsbereichen
 - Siehe Kontextanalyse
 - Öffne Geltungsbereich bei Anhängen von Block an Pfad
 - Schließe Geltungsbereich bei Entfernen von Block aus Pfad

Beispiel Superlokale VN 1 im SSA-CFG



Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier $a + b$ unbekannt
- $e + f$ berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer. z.B.
 - Verändern bereits bearbeiteter Blöcke

Regionale Methoden

- Superlokale VN verwirft gesamte Tabelle bei Merge Point
 - Nächste Verfeinerung: Über Merge Points hinweg arbeiten
- ➔ **Regionaler** Anwendungsbereich
- Größer als superlokal
 - Kleiner als gesamte Prozedur

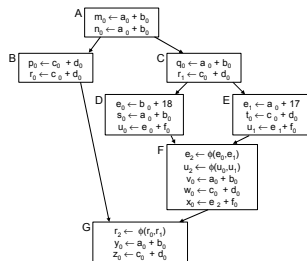
Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
 - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
 - Welche Einträge streichen? (z.B. $\mathbf{b+18}$ und $\mathbf{a+17}$)
 - Liegen auf unterschiedlichen Pfaden
 - Gleiche Ausdrücke in beiden Pfaden
 - Aber mit unterschiedlichen Wertnummern ($\mathbf{e+f}$)
 - **Kompliziert** und rechenintensiv

Anderer Ansatz: Beschränkte Tabelle auf Fakten, die **unabhängig** vom konkreten Pfad gelten

A. Koch

- Beide Pfade zu F haben gleichen **Präfix** (A,C)
- Alle Operationen in A und C sind **immer** vor Erreichen von F ausgeführt worden
- Letzter “sicherer” Stand ist also Tabelle nach C
- Als Ausgangspunkt für F benutzen



Wie mit Zuweisungen in D und E umgehen?

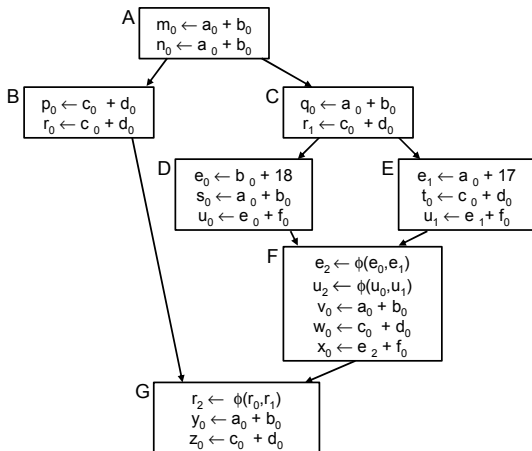
- SSA-Form umgeht Verlorengehen von Werten
 - Eigene Variable für jeden Wert
 - D und E können **zusätzliche** Werte erzeugen
 - Aber können **keine** alten Werte streichen
 - Konflikte werden über ϕ -Funktionen aufgelöst

Noch größere Anwendungsbereiche 5

Effekt der Vorgehensweise: Verwende C als Ausgangspunkt für F

A. Koch

- Erkennt jetzt Redundanz von $a_0 + b_0$ und $c_0 + d_0$ in F
- Verpasst aber $e_2 + f_0$, da **zwischen** C und F berechnet



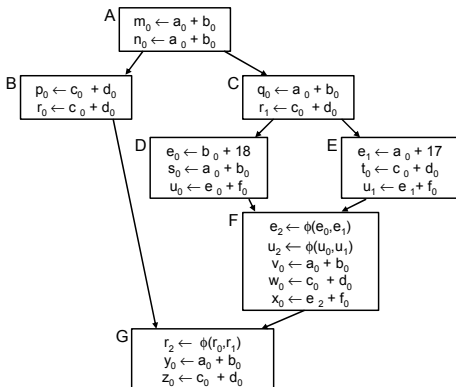
Noch größere Anwendungsbereiche 6

Verfahren benötigt: Letzten gemeinsamen Vorfahren über alle Pfade zu einem Block

Benutze Tabelle von X bei Eintritt in Y: $X \leftarrow Y$

A. Koch

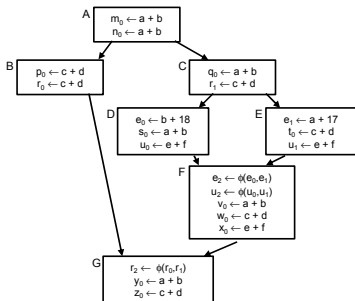
- $- \leftarrow A$
- $A \leftarrow B$
- $A \leftarrow C$
- $C \leftarrow D$
- $C \leftarrow E$
- $C \leftarrow F$
- $A \leftarrow G$



➔ X muß Y dominieren

Dominatormengen

Block	Dom	IDom
A	A	-
B	A,B	A
C	A,C	A
D	A,C,D	C
E	A,C,E	C
F	A,C,F	C
G	A,G	A



- Geben sei ein Block b
- Jeder Block d in $\text{DOM}(b) - \{b\}$ wurde **vor** b ausgeführt
- Die VN-Tabelle jedes d **könnte** zur Vorbelegung von b verwendet werden
- Welche wäre die beste Wahl?
- Die von Block $e = \text{IDOM}(b)$!
 - e wird von allen anderen Blöcken aus $\text{DOM}(b) - \{b\}$ dominiert
 - Kann all deren Informationen verwenden
 - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)

Regionaler Algorithmus

A. Koch

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
 - Verschachtelte Hash-Tabellen für Geltungsbereiche
 - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten x
 - Mit Tabelle von $IDOM(x)$
 - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
 - Constant folding
 - Ausnutzung algebraischer Eigenschaften

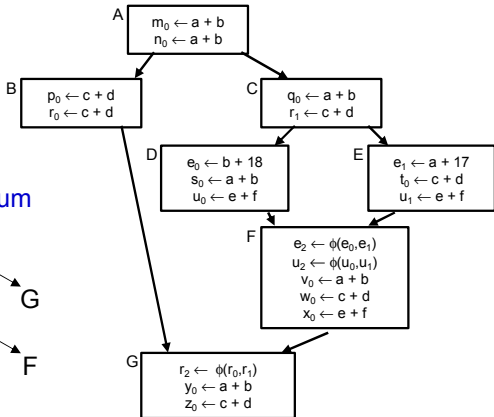
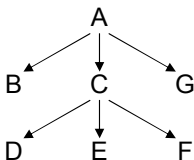
➔ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen

Arbeitet in Prefix-Reihenfolge auf **Dominatorbaum**

- Stellt sicher, das $IDOM(x)$ vor x bearbeitet ist

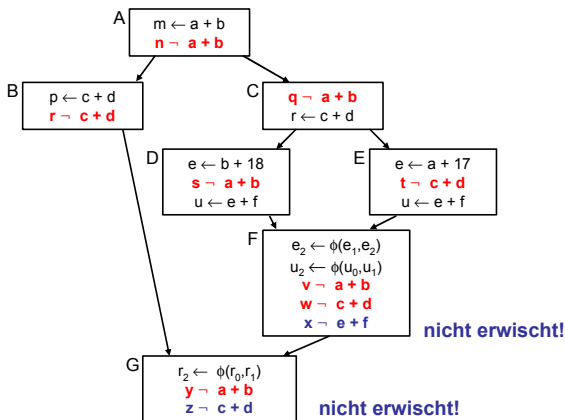
A. Koch

Dominatorbaum



Mögliche Reihenfolge: A, B, G, C, D, E, F

Beispiel DVNT 3



- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen
 - Rückwärtskanten im CFG

Globale Methoden

- Über CFG der **kompletten** Prozedur
- Allgemeine Vorgehensweise: Schritte trennen
- **Analyse** sammelt Informationen, auch über Zyklus hinweg
- **Transformation** erst, wenn alles fertig analysiert

Am Beispiel **Global Common Subexpression Elimination**

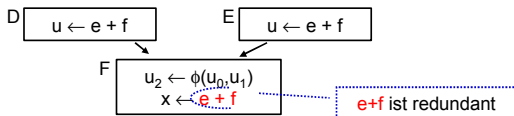
- Nun wieder **lexikalischer** Vergleich von **Namen**
- **Keine** Wertnummern mehr
- Keine SSA-Form mehr
- Demonstriert das wichtige Mittel der **Datenflussanalyse**

- **Global Value Numbering**
- Graphenbasierte Vorgehensweise
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
 - ... aus Zeitmangel
 - Siehe Muchnick Abschnitt 13.1.2

- Eine Auswertung eines Ausdrucks e an der Stelle p ist redundant
- ... genau dann, wenn
 - Jeder Pfad vom Prozedurstart zu p enthält eine Auswertung von e
 - **und** die Werte der Teilausdrücke von e haben sich von dort zu p nicht geändert

A. Koch

➔ Auswertung e an p liefert **gleichen** Wert wie frühere Auswertung(en)



Wie diese redundanten (Teil)ausdrücke finden?

Definition

Ein Ausdruck e ist **definiert** an einer Stelle p im CFG falls sein Wert an der Stelle p berechnet wird. p ist damit eine **Definitionsstelle** von e .

Auslöschung

Ein Ausdruck e wird **ausgelöscht** an einer Stelle p im CFG falls ein oder mehrere seiner Operanden an der Stelle p definiert werden. p ist damit eine **Auslöschungsstelle** von e .

Verfügbarkeit

Ein Ausdruck e ist **verfügbar** an einer Stelle p wenn jeder zu p führende Pfad im CFG, beginnend beim Prozeduranfang, eine vorhergehende Definition von e enthält und e nicht zwischen dieser Definitionsstelle und p ausgelöscht wird.

Hier lexikalisch, **nicht** mehr über Wertnummern!

A. Koch

```
x := e + f; // Definition von e+f  
e := 5;    // Ausloeschung e+f  
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
 - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
 - Operanden**namen** und Operatoren
 - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
 - Minimal: Anzahl Variablen plus Anzahl Konstanten
 - Maximal: Anzahl von (Teil)Ausdrücken im CFG

Ziel

Wenn ein Ausdruck e in einem Block **verfügbar** ist, braucht er dort nicht neu berechnet zu werden.

... nun über Basis-Blockgrenzen und Verzweigungen hinweg!

1. Schritt: Analyse

- Formuliere **Gleichungssystem** über den CFG der Prozedur
- Löse Gleichungssystem, um Menge **verfügbarer** Ausdrücke zu bestimmen

$AVAIL(b)$

$AVAIL(b)$ sei Menge der am Anfang von Block b verfügbaren Ausdrücke.

Sicherheit

A. Koch

- $x + y \in \text{AVAIL}(b)$ **beweist**, dass eine **vorherige** Auswertung von $x + y$ existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
 - ... auf verschiedene Weisen realisierbar

Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein
 - Im Prinzip billig
 - Viele können auch entfernt werden
 - Haben aber Einfluß auf Lebenszeiten (\rightarrow *Live*)

Berechnung von $AVAIL(b)$ 1

Über zwei Hilfsfunktionen

$EXPRKILL(b)$

Die Menge der bis zum Ende des Blocks b ausgelöschten Ausdrücke.

$DEEXPR(b)$ (*downward exposed*)

Menge der im Block b definierten Ausdrücke, die bis zum Ende des Blocks nicht ausgelöscht werden.

Weiterleitung von Ausdrücken:

- Wenn Ausdruck e bei Eintritt in Block b verfügbar ist, und **nicht** in $EXPRKILL(b)$ ist, dann ist e auch nach Ende von b verfügbar.

Berechnung von $AVAIL(b)$ 2

Damit nun definierbar

A. Koch

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

$$AVAIL(b_0) = \emptyset$$

mit:

$\text{pred}(b)$: Vorgängerblöcke von b im CFG

b_0 Startblock des CFG

➔ Datenflußproblem, lösen mit Standardverfahren

Vor den Details erstmal einen Schritt zurück: Wie soll es weitergehen?

- 1 Berechne $AVAIL(b)$ für alle Blöcke b
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in $AVAIL(b)$
- 3 Innerhalb der Blöcke b dann **lokales Value Numbering**
 - Initialisiert block-lokale Tabelle mit Ausdrücken aus $AVAIL(b)$

Berechnung von $AVAIL(b)$ 3

Baut auf Berechnung von $DEEXPR$ und $EXPRKILL$ auf

assume a block b with operations o_1, o_2, \dots, o_k

$VARKILL \leftarrow \emptyset$

$DEEXPR(b) \leftarrow \emptyset$

for $i = k$ **to** 1

 assume o_i is " $x \rightarrow y + z$ "

 add x to $VARKILL$

if ($y \notin VARKILL$) **and** ($z \notin VARKILL$) **then**

 add " $y + z$ " to $DEEXPR(b)$

$EXPRKILL(b) \leftarrow \emptyset$

For each expression e

for each variable $v \in e$

if $v \in VARKILL(b)$ **then**

$EXPRKILL(b) \leftarrow EXPRKILL(b) \cup \{e\}$

assume a block b with operations o_1, o_2

$VARKILL \leftarrow \emptyset$

$DEEXPR(b) \leftarrow \emptyset$

Rückwärts durch

for $i = k$ **to** 1

 assume o_i is " $x \rightarrow y + z$ "

 add x to $VARKILL$

if ($y \notin VARKILL$) **and** ($z \notin VARKILL$) **then**

 add " $y + z$ " to $DEEXPR(b)$

$EXPRKILL(b) \leftarrow \emptyset$

For each expression e

for each variable $v \in e$

if $v \in VARKILL(b)$ **then**

$EXPRKILL(b) \leftarrow EXPRKILL(b) \cup \{e\}$

- “**Foreach** expression e ”
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
 - Hash-Map M von $v \rightarrow E$ bildet Variable ab auf benutzende Ausdrücke
 - Dann mit v über $VARKILL(b)$ iterieren
 - Für jede Variable v via $M(v)$ Ausdrücke E bestimmen
 - ... und E in $EXPRKILL(b)$ aufnehmen

Berechnung von $AVAIL(b)$ 5

Nun Anwendung eines iterativen Algorithmus zum Finden eines Fixpunktes

$Worklist \leftarrow \{ \text{all blocks in CFG} \}$

while ($Worklist \neq \emptyset$)

 remove a block b from $Worklist$

 recompute $AVAIL(b)$ as

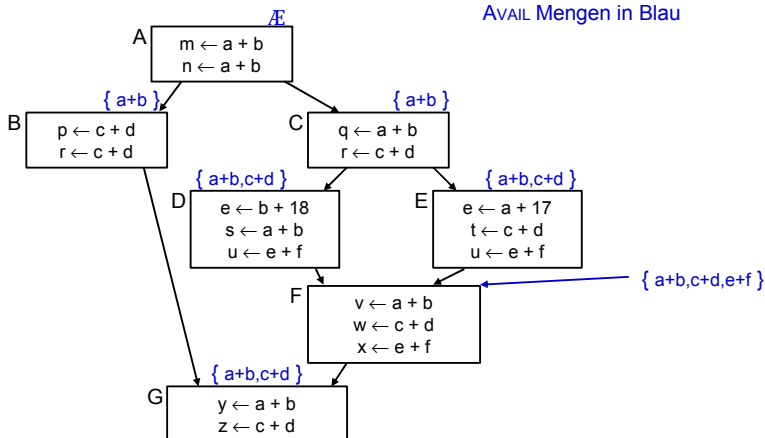
$$AVAIL(b_i) = \bigcap_{x \in \text{pred}(b)} (DEEXPR(x) \cup (AVAIL(x) \cap \overline{EXPRKILL(x)}))$$

if $AVAIL(b)$ changed **then**

$Worklist \leftarrow Worklist \cup \text{successors}(b)$

Beweis der Terminierung: Später ...

Beispiel mit $AVAIL(b)$ -Mengen

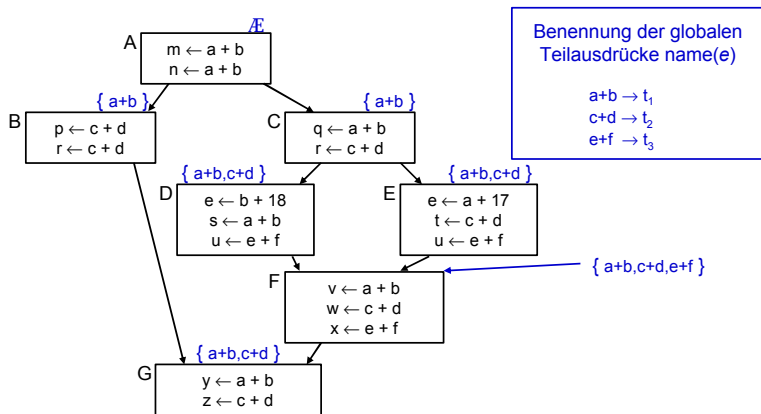


- 1 Berechne $AVAIL(b)$ für alle Blöcke b
 - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in $AVAIL(b)$
- 3 Innerhalb der Blöcke b dann **lokales Value Numbering**
 - Initialisiert block-lokale Tabelle mit Ausdrücken aus $AVAIL(b)$

Nun eindeutige Namen t_i für global bekannte (Teil-)ausdrücke e_i vergeben

➡ Hashing über Ausdrücke und Durchnummerieren mit i

Beispiel mit benannten globalen CSEs



Damit jetzt Schritt 3 (Transformation) vornehmen!

Block-lokales Value Numbering in Block b

1. Phase: Finde Wiederbenutzungen eines Ausdrucks

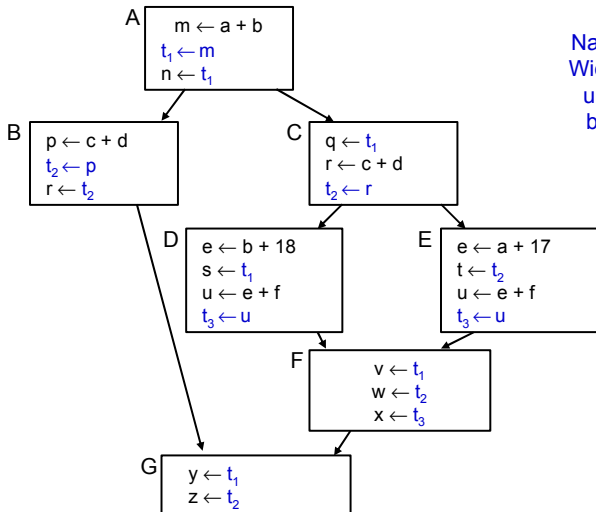
- 1 Initialisiere Hash-Tabelle mit $AVAIL(b)$
 - Falls VN Versionsnummern benutzt:
Ausdrücke aus $AVAIL(b)$ umformen
z.B. $e + f \rightarrow e_0 + f_0$
- 2 Wenn Wiederverwendung $\mathbf{x} := e_i$ erkannt
 - Ersetze e durch Verweis auf bekannten Namen t_i :
 $\mathbf{x} := t_i$
 - Merke Wiederverwendung durch $USED[e] := true$

2. Phase: Kopien der benutzten Werte unter bekanntem Namen erzeugen

- Für alle Ausdrücke e im Basisblock b
 - Wenn $e \in \text{DEEXPR}(b)$ und $\text{USED}[e]$
 - Füge nach letzter Definition von e in b ein: $t_i := e$

- **Lokale** Redundanzen durch lokales VN beseitigt
- **Globale** Redundanzen durch AVAIL-Mengen beseitigt
- Nicht ganz identischer Effekt, findet
 - Lokale Redundanzen durch Wertgleichheit
 - Globale Redundanzen durch gleiche Schreibweise

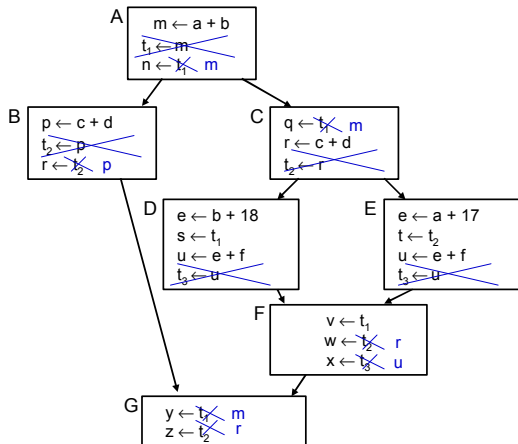
Effekt von globalem CSE im Beispiel



Nach Auffinden von
Wiederbenutzungen
und Kopieren der
benutzten Werte

Unnötige Kopien

A. Koch

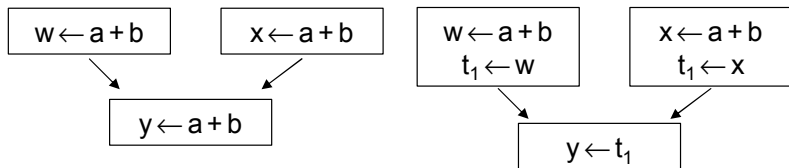


Viele der Kopien unnötig,
hier sogar **alle**.

Können später aber entfernt
werden (copy propagation,
copy coalescing)

Hier wird Kopie gebraucht:

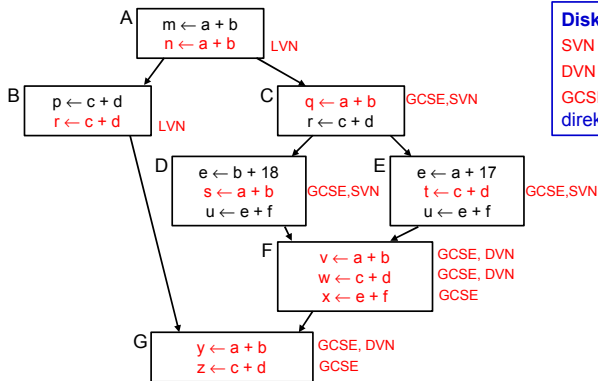
A. Koch



hier kann nicht "w oder x"
geschrieben werden

Unterschied zu letztem Beispiel:
In beiden Zweigen nur **eine** Variable (u).

Vergleich der Methoden 1



Diskussion:

SVN umfasst LVN

DVN umfasst SVN

GCSE & xVN sind nicht
direkt vergleichbar

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:
 $(a + b) = (c + d)$, wenn $a = d$ und $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
 - Verwendung von LVN innerhalb von Basisblöcken

- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen (\rightarrow SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken