

Optimierende Compiler

Skalare Optimierung 1

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2009

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

OptComp

A. Koch

Organisatorisches

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

Organisatorisches

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- 1. Teilprüfung vom 3.6. auf 17.6. verschoben
- Individuelle Prüfungszeiten unverändert
- Stoffpensum: Nur der bis zum 3.6. behandelte Stoff

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- 1. Teilprüfung vom 3.6. auf 17.6. verschoben
- Individuelle Prüfungszeiten unverändert
- Stoffpensum: Nur der bis zum 3.6. behandelte Stoff

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- 1. Teilprüfung vom 3.6. auf 17.6. verschoben
- Individuelle Prüfungszeiten unverändert
- Stoffpensum: Nur der bis zum 3.6. behandelte Stoff

Dead Code Elimination

- Nutzloser Code

- Keine weitere Operation verwendet Ergebnis
- Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar

- Unerreichbarer Code

- Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes
Dead Code Elimination

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Nutzloser Code
 - Keine weitere Operation verwendet Ergebnis
 - Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar
- Unerreichbarer Code
 - Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes
Dead Code Elimination

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Nutzloser Code
 - Keine weitere Operation verwendet Ergebnis
 - Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar
- Unerreichbarer Code
 - Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes
Dead Code Elimination

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Nutzloser Code
 - Keine weitere Operation verwendet Ergebnis
 - Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar
- Unerreichbarer Code
 - Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes
Dead Code Elimination

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Nutzloser Code
 - Keine weitere Operation verwendet Ergebnis
 - Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar
- Unerreichbarer Code
 - Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes
Dead Code Elimination

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Nutzloser Code
 - Keine weitere Operation verwendet Ergebnis
 - Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar
- Unerreichbarer Code
 - Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes
Dead Code Elimination

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Nutzloser Code
 - Keine weitere Operation verwendet Ergebnis
 - Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar
- Unerreichbarer Code
 - Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes
Dead Code Elimination

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➔ Klassisches *Mark-and-Sweep* Vorgehen

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➔ Klassisches *Mark-and-Sweep* Vorgehen

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➔ Klassisches *Mark-and-Sweep* Vorgehen

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➔ Klassisches *Mark-and-Sweep* Vorgehen

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➔ Klassisches *Mark-and-Sweep* Vorgehen

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➡ Klassisches *Mark-and-Sweep* Vorgehen

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➡ Klassisches *Mark-and-Sweep* Vorgehen

- Markieren benötigter Operationen
 - Markiere kritische Operationen
 - Untersucht deren Operanden und markiert die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➔ Klassisches *Mark-and-Sweep* Vorgehen

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

Neue Konzepte erforderlich!

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

Neue Konzepte erforderlich!

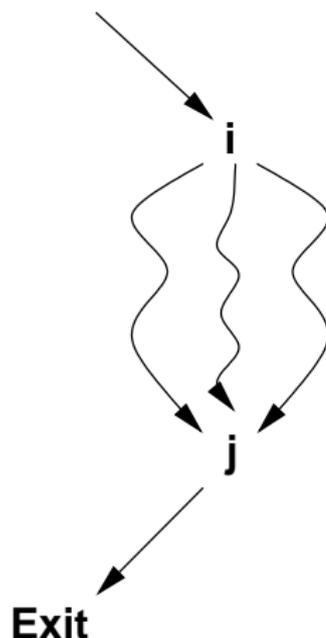
- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

Neue Konzepte erforderlich!

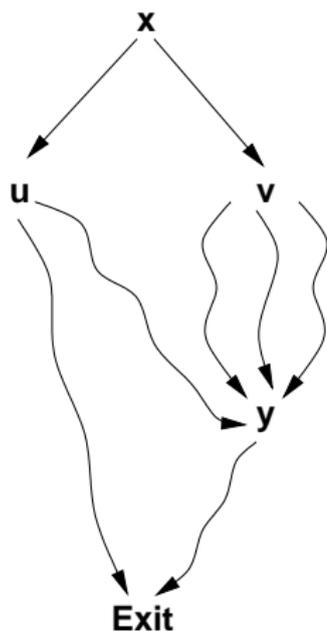
Ein Knoten j **postdominiert** den Knoten i in einem CFG, wenn alle Pfade von i zum Endknoten des CFG durch den Knoten j führen.



y ist von x **kontrollabhängig** genau dann, wenn

- 1 es einen nicht-leeren Pfad von x zu y gibt und jeder auf diesem Pfad liegende Knoten von y postdominiert wird,
- 2 x aber nicht strikt von y postdominiert wird.

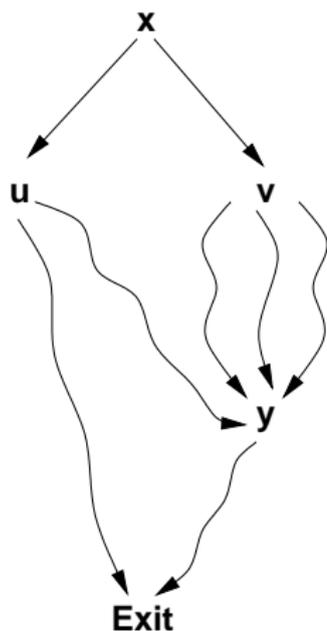
- y postdominiert v und alle Knoten dazwischen
- y postdominiert nicht x
- y ist von x **kontrollabhängig**



y ist von x **kontrollabhängig** genau dann, wenn

- 1 es einen nicht-leeren Pfad von x zu y gibt und jeder auf diesem Pfad liegende Knoten von y postdominiert wird,
- 2 x aber nicht strikt von y postdominiert wird.

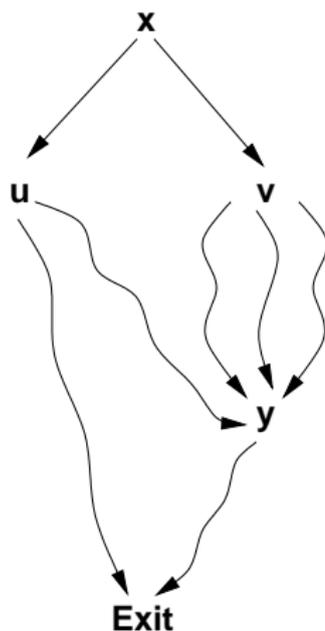
- y postdominiert v und alle Knoten dazwischen
- y postdominiert nicht x
- y ist von x **kontrollabhängig**



y ist von x **kontrollabhängig** genau dann, wenn

- 1 es einen nicht-leeren Pfad von x zu y gibt und jeder auf diesem Pfad liegende Knoten von y postdominiert wird,
- 2 x aber nicht strikt von y postdominiert wird.

- y postdominiert v und alle Knoten dazwischen
- y postdominiert nicht x
- y ist von x **kontrollabhängig**



Andere Deutung

- Zwei oder mehr Kanten verlassen Block x
- Nach Eintritt in eine der Kanten wird y in jeden Fall ausgeführt
- Über die andere(n) Kante(n) kann der Endknoten ohne y erreicht werden

Damit entscheidet Bedingung am Ende von x , ob y ausgeführt wird.

➡ Wenn Anweisung in y benötigt wird, wird damit auch die Entscheidung in x benötigt

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Andere Deutung

- Zwei oder mehr Kanten verlassen Block x
- Nach Eintritt in eine der Kanten wird y in jeden Fall ausgeführt
- Über die andere(n) Kante(n) kann der Endknoten ohne y erreicht werden

Damit entscheidet Bedingung am Ende von x , ob y ausgeführt wird.

➡ Wenn Anweisung in y benötigt wird, wird damit auch die Entscheidung in x benötigt

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Andere Deutung

- Zwei oder mehr Kanten verlassen Block x
- Nach Eintritt in eine der Kanten wird y in jeden Fall ausgeführt
- Über die andere(n) Kante(n) kann der Endknoten ohne y erreicht werden

Damit entscheidet Bedingung am Ende von x , ob y ausgeführt wird.

➡ Wenn Anweisung in y benötigt wird, wird damit auch die Entscheidung in x benötigt

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Andere Deutung

- Zwei oder mehr Kanten verlassen Block x
- Nach Eintritt in eine der Kanten wird y in jeden Fall ausgeführt
- Über die andere(n) Kante(n) kann der Endknoten ohne y erreicht werden

Damit entscheidet Bedingung am Ende von x , ob y ausgeführt wird.

➡ Wenn Anweisung in y benötigt wird, wird damit auch die Entscheidung in x benötigt

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Andere Deutung

- Zwei oder mehr Kanten verlassen Block x
- Nach Eintritt in eine der Kanten wird y in jeden Fall ausgeführt
- Über die andere(n) Kante(n) kann der Endknoten ohne y erreicht werden

Damit entscheidet Bedingung am Ende von x , ob y ausgeführt wird.

➡ Wenn Anweisung in y benötigt wird, wird damit auch die Entscheidung in x benötigt

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Andere Deutung

- Zwei oder mehr Kanten verlassen Block x
- Nach Eintritt in eine der Kanten wird y in jeden Fall ausgeführt
- Über die andere(n) Kante(n) kann der Endknoten ohne y erreicht werden

Damit entscheidet Bedingung am Ende von x , ob y ausgeführt wird.

➡ Wenn Anweisung in y benötigt wird, wird damit auch die Entscheidung in x benötigt

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten verkehrt
 - umgekehrter CFG (uCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der y
nahegelegenste Punkt, an dem die Entscheidung fällt?

➔ Wo ist der y nahegelegenste Knoten, bei dem auch eine
Abzweigung an y vorbei genommen werden kann?

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der y
nahegelegenste Punkt, an dem die Entscheidung fällt?

➔ Wo ist der y nahegelegenste Knoten, bei dem auch eine
Abzweigung an y vorbei genommen werden kann?

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der *y* **nahegelegenste** Punkt, an dem die Entscheidung fällt?

➔ Wo ist der *y* nahegelegenste Knoten, bei dem auch eine Abzweigung an *y* vorbei genommen werden kann?

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der *y* **nahegelegenste** Punkt, an dem die Entscheidung fällt?

➔ Wo ist der *y* nahegelegenste Knoten, bei dem auch eine Abzweigung an *y* vorbei genommen werden kann?

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der y **nahegelegenste** Punkt, an dem die Entscheidung fällt?

➔ Wo ist der y nahegelegenste Knoten, bei dem auch eine Abzweigung an y vorbei genommen werden kann?

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der y
nahegelegenste Punkt, an dem die Entscheidung fällt?

➔ Wo ist der y nahegelegenste Knoten, bei dem auch eine
Abzweigung an y vorbei genommen werden kann?

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der y
nahegelegenste Punkt, an dem die Entscheidung fällt?

➔ Wo ist der y nahegelegenste Knoten, bei dem auch eine
Abzweigung an y vorbei genommen werden kann?

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

Hat etwas mit **Postdominanz** zu tun.

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im umgekehrten CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der y **nahegelegenste** Punkt, an dem die Entscheidung fällt?

➔ Wo ist der y nahegelegenste Knoten, bei dem auch eine Abzweigung an y vorbei genommen werden kann?

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

Analoge Betrachtung bei Dominatoren:
Welche Knoten w liegen gerade außerhalb der Dominanz
eines Knotens x ?

Anders: An welchem Knoten w kann aus dessen
Ausführung nicht mehr sicher auf die Ausführung von x
geschlossen werden?

Dominatorgrenze $DF(x)$

Knoten w , bei denen ein Vorgänger q durch x dominiert wird
($q \in \text{preds}(w) \wedge x \in \text{DOM}(q)$), aber w selbst nicht von x
strikt dominiert ist ($x \notin \text{DOM}(w) - \{w\}$), heissen die
Dominatorgrenze von x , mit $w \in DF(x)$.

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

Analoge Betrachtung bei Dominatoren:
Welche Knoten w liegen gerade außerhalb der Dominanz
eines Knotens x ?

Anders: An welchem Knoten w kann aus dessen
Ausführung nicht mehr sicher auf die Ausführung von x
geschlossen werden?

Dominatorgrenze $DF(x)$

Knoten w , bei denen ein Vorgänger q durch x dominiert wird
($q \in \text{preds}(w) \wedge x \in \text{DOM}(q)$), aber w selbst nicht von x
strikt dominiert ist ($x \notin \text{DOM}(w) - \{w\}$), heissen die
Dominatorgrenze von x , mit $w \in DF(x)$.

OptComp

A. Koch

Organisatorisch

Dead Code
EliminationBereinigen
des CFG

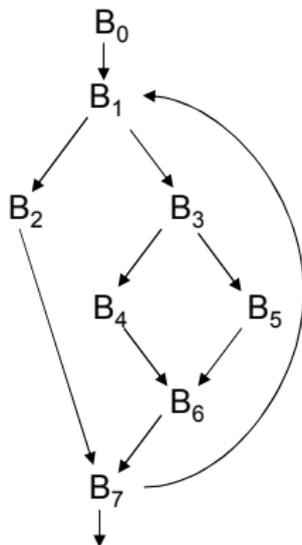
Zusammenfassu

Analoge Betrachtung bei Dominatoren:
Welche Knoten w liegen gerade außerhalb der Dominanz
eines Knotens x ?

Anders: An welchem Knoten w kann aus dessen
Ausführung nicht mehr sicher auf die Ausführung von x
geschlossen werden?

Dominatorgrenze $DF(x)$

Knoten w , bei denen ein Vorgänger q durch x dominiert wird
($q \in \text{preds}(w) \wedge x \in \text{DOM}(q)$), aber w selbst nicht von x
strikt dominiert ist ($x \notin \text{DOM}(w) - \{w\}$), heissen die
Dominatorgrenze von x , mit $w \in DF(x)$.



	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

OptComp

A. Koch

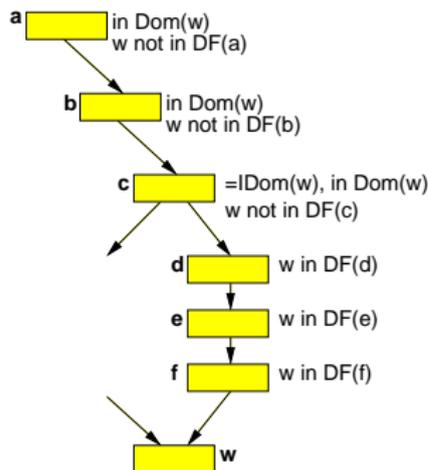
Organisatorisch

Dead Code
Elimination

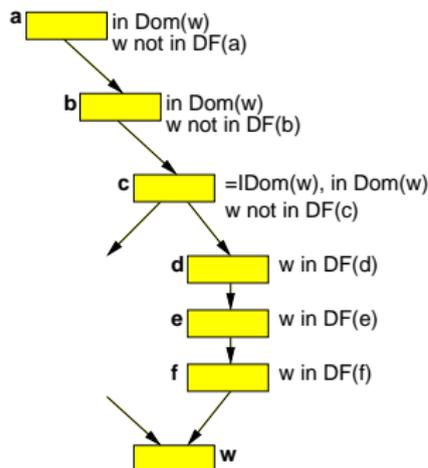
Bereinigen
des CFG

Zusammenfass

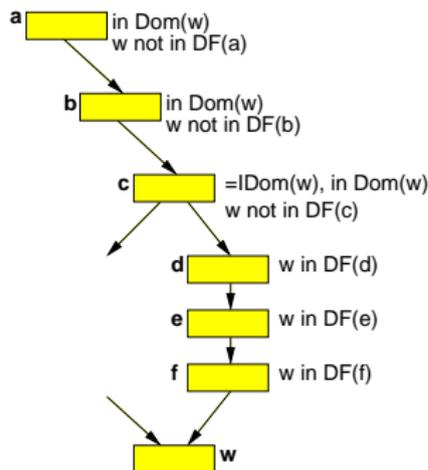
- Knoten auf Dominatorgrenze sind immer Merge-Knoten
- Vorgänger x eines Merge-Knotens w haben $w \in DF(x)$, wenn nicht gilt $x \in \text{DOM}(w)$
- Dominatoren z der Vorgänger x von w haben auch $w \in DF(z)$, wenn nicht gilt $z \in \text{DOM}(w)$



- Knoten auf Dominatorgrenze sind immer Merge-Knoten
- Vorgänger x eines Merge-Knotens w haben $w \in DF(x)$, wenn nicht gilt $x \in \text{DOM}(w)$
- Dominatoren z der Vorgänger x von w haben auch $w \in DF(z)$, wenn nicht gilt $z \in \text{DOM}(w)$

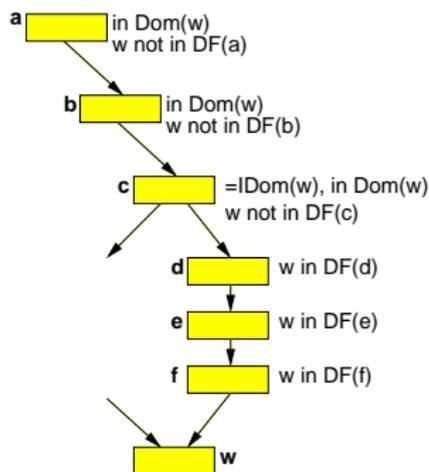


- Knoten auf Dominatorgrenze sind immer Merge-Knoten
- Vorgänger x eines Merge-Knotens w haben $w \in DF(x)$, wenn nicht gilt $x \in \text{DOM}(w)$
- Dominatoren z der Vorgänger x von w haben auch $w \in DF(z)$, wenn nicht gilt $z \in \text{DOM}(w)$



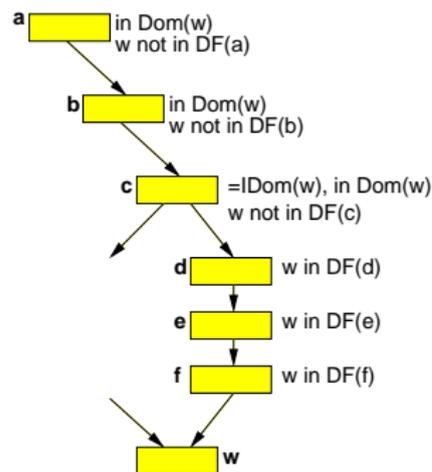
Berechnung von Dominatorgrenzen - Vorgehensweise

- 1 Finde Merge-Points als w
- 2 Beginne Untersuchung bei direkten Vorgängern x des Merge-Points w
- 3 Klettere rückwärts weiter via IDOM des aktuellen Knotens x
 - Setze $w \in DF(x)$, bis $x = IDOM(w)$



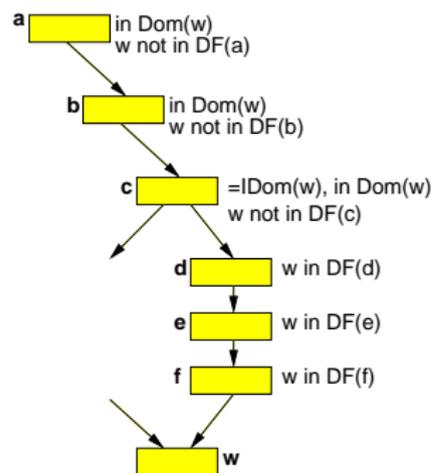
Berechnung von Dominatorgrenzen - Vorgehensweise

- 1 Finde Merge-Points als w
- 2 Beginne Untersuchung bei direkten Vorgängern x des Merge-Points w
- 3 Klettere rückwärts weiter via IDOM des aktuellen Knotens x
 - Setze $w \in DF(x)$, bis $x = IDOM(w)$



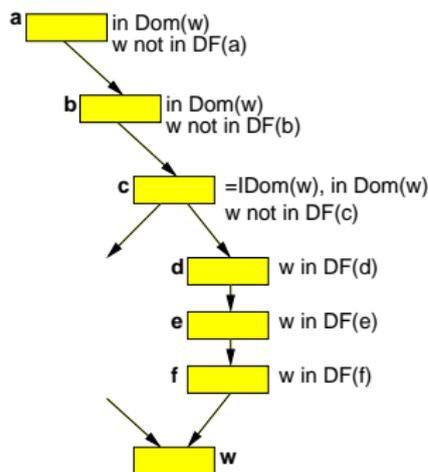
Berechnung von Dominatorgrenzen - Vorgehensweise

- 1 Finde Merge-Points als w
- 2 Beginne Untersuchung bei direkten Vorgängern x des Merge-Points w
- 3 Klettere rückwärts weiter via IDOM des aktuellen Knotens x
 - Setze $w \in DF(x)$, bis $x = IDOM(w)$



Berechnung von Dominatorgrenzen - Vorgehensweise

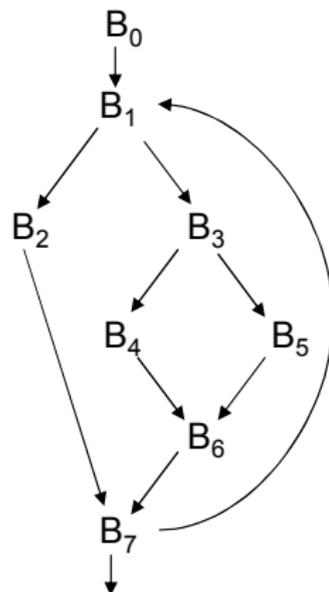
- 1 Finde Merge-Points als w
- 2 Beginne Untersuchung bei direkten Vorgängern x des Merge-Points w
- 3 Klettere rückwärts weiter via IDOM des aktuellen Knotens x
 - Setze $w \in DF(x)$, bis $x = \text{IDOM}(w)$



Berechnung von Dominatorgrenzen - Algorithmus

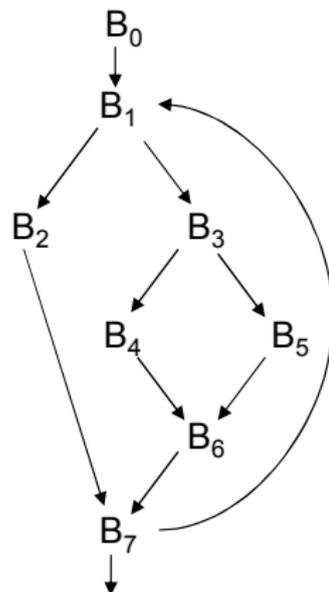
```
foreach node n in CFG do  
  DF(n) :=  $\emptyset$   
foreach node n in CFG do  
  if |preds(n)| > 1 then  
    foreach p in preds(n) do  
      runner := p  
      while runner  $\neq$  IDOM(n)  $\wedge$  runner  $\neq$  n do  
        DF(runner) := DF(runner)  $\cup$  { n }  
        runner := IDOM(runner)
```

- Bearbeite B6: Zu B5, dort B6 in DF(B5), Ende bei B3. Zu B4, dort B6 in DF(B4), Ende bei B3.
- Bearbeite B7: Zu B2, dort B7 in DF(B2), Ende bei B1. Zu B6, dort B7 in DF(B6), zu B3, dort B7 in DF(B3), Ende bei B1.
- Bearbeite B1: Zu B0, dort Ende. Zu B7, dort B1 in DF(B7), dort Ende.



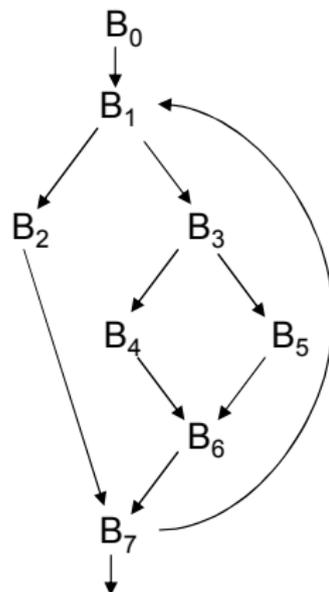
	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

- Bearbeite B6: Zu B5, dort B6 in DF(B5), Ende bei B3. Zu B4, dort B6 in DF(B4), Ende bei B3.
- Bearbeite B7: Zu B2, dort B7 in DF(B2), Ende bei B1. Zu B6, dort B7 in DF(B6), zu B3, dort B7 in DF(B3), Ende bei B1.
- Bearbeite B1: Zu B0, dort Ende. Zu B7, dort B1 in DF(B7), dort Ende.



	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

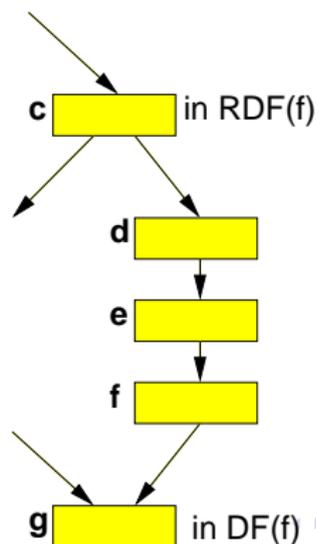
- Bearbeite B6: Zu B5, dort B6 in DF(B5), Ende bei B3. Zu B4, dort B6 in DF(B4), Ende bei B3.
- Bearbeite B7: Zu B2, dort B7 in DF(B2), Ende bei B1. Zu B6, dort B7 in DF(B6), zu B3, dort B7 in DF(B3), Ende bei B1.
- Bearbeite B1: Zu B0, dort B1 in DF(B0), Ende. Zu B7, dort B1 in DF(B7), dort Ende.



	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

Gesucht: Verzweigungen, von denen benötigte Anweisung i **kontrollabhängig** ist

➔ Sind Dominatorgrenzen von $\text{block}(i)$ im reversen CFG:
RDF(block(i))



MarkPass

```
foreach op i
  clear i's mark
  if i is critical then
    mark i
    add i to WorkList
while (Worklist  $\neq$   $\emptyset$ )
  remove i from WorkList
    (i has form "x  $\rightarrow$  y op z")
  if def(y) is not marked then
    mark def(y)
    add def(y) to WorkList
  if def(z) is not marked then
    mark def(z)
    add def(z) to WorkList
  foreach b  $\in$  RDF(block(i))
    mark the block-ending
      branch j in b
    add j to WorkList
```

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

```
Sweep
  foreach op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to i's nearest useful post-dominator
      if i is not a jump then
        delete i
```

- Lösche unmarkierte Operationen
- “Verbiege” unmarkierte Verzweigung
 - Setze Ausführung bei nächstgelegenen Postdominator mit nützlichen Operationen fort

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

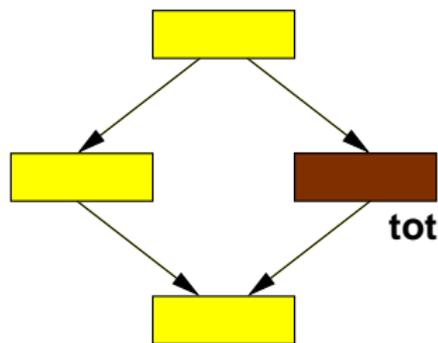
```
Sweep
  foreach op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to i's nearest useful post-dominator
      if i is not a jump then
        delete i
```

- Lösche unmarkierte Operationen
- “Verbiege” unmarkierte Verzweigung
 - Setze Ausführung bei nächstgelegenen Postdominator mit nützlichen Operationen fort

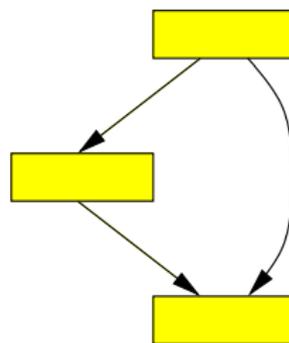
```
Sweep
  foreach op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to i's nearest useful post-dominator
      if i is not a jump then
        delete i
```

- Lösche unmarkierte Operationen
- “Verbiege” unmarkierte Verzweigung
 - Setze Ausführung bei nächstgelegenen Postdominator mit nützlichen Operationen fort

Beispiel Verbiegen von Verzweigungen 1

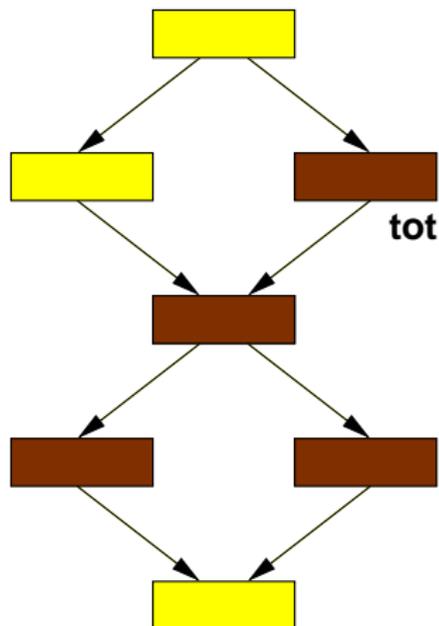


Vorher

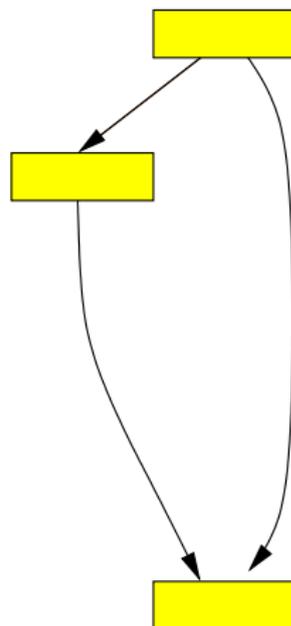


Nachher

Beispiel Verbiegen von Verzweigungen 2



Vorher



Nachher

- **Gesamter Ablauf von Dead():**
 - 1 MarkPass()
 - 2 SweepPass()
- Kann leere Blöcke hinterlassen
- Aufräumen mit nächstem Algorithmus

- Gesamter Ablauf von Dead():
 - 1 MarkPass()
 - 2 SweepPass()
- Kann leere Blöcke hinterlassen
- Aufräumen mit nächstem Algorithmus

- Gesamter Ablauf von Dead():
 - 1 MarkPass()
 - 2 SweepPass()
- Kann leere Blöcke hinterlassen
- Aufräumen mit nächstem Algorithmus

- Gesamter Ablauf von Dead():
 - 1 MarkPass()
 - 2 SweepPass()
- Kann leere Blöcke hinterlassen
- Aufräumen mit nächstem Algorithmus

- Gesamter Ablauf von Dead():
 - 1 MarkPass()
 - 2 SweepPass()
- Kann leere Blöcke hinterlassen
- Aufräumen mit nächstem Algorithmus

Bereinigen des CFG

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➔ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➔ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➔ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➡ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➔ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➔ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➔ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

➔ Algorithmus CLEAN: Vier Schritte

OptComp

A. Koch

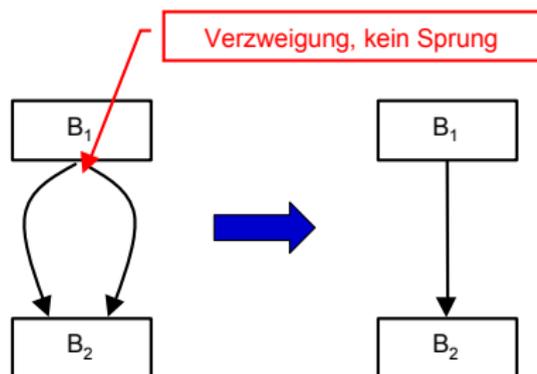
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

- Entsteht durch:
Verbiegen von
Verzweigungen
- Vorgehen: Ersetze
Verzweigung durch
Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

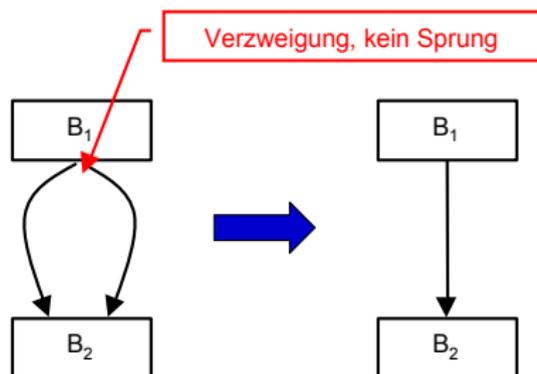
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

- Entsteht durch:
Verbiegen von
Verzweigungen
- Vorgehen: Ersetze
Verzweigung durch
Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

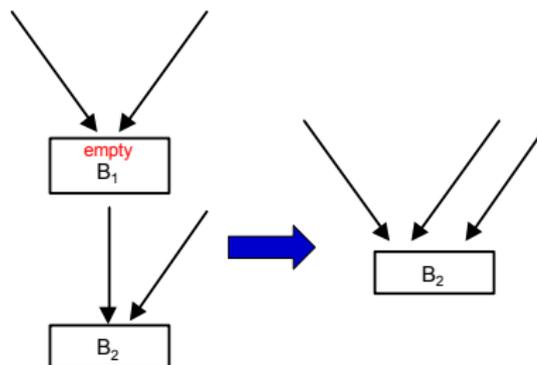
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch:
Gelöschte Operationen
in B1
- Voraussetzungen:
Leerer Block B1 endet
mit Sprung
- Vorgehen:
 - Verbiege eingehende
Kanten von B1 zu B2
 - Entferne B1



Quelle: C&T Fig. 10.4

OptComp

A. Koch

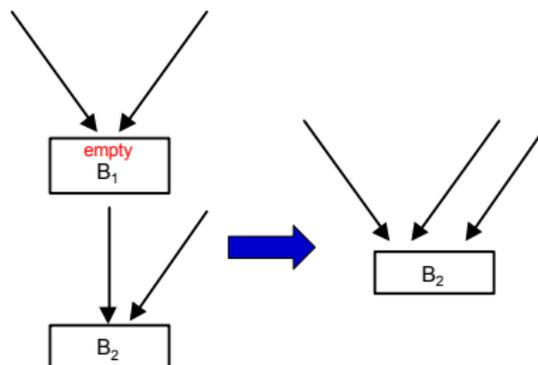
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch:
Gelöschte Operationen
in B1
- Voraussetzungen:
Leerer Block B1 endet
mit Sprung
- Vorgehen:
 - Verbiege eingehende
Kanten von B1 zu B2
 - Entferne B1



Quelle: C&T Fig. 10.4

OptComp

A. Koch

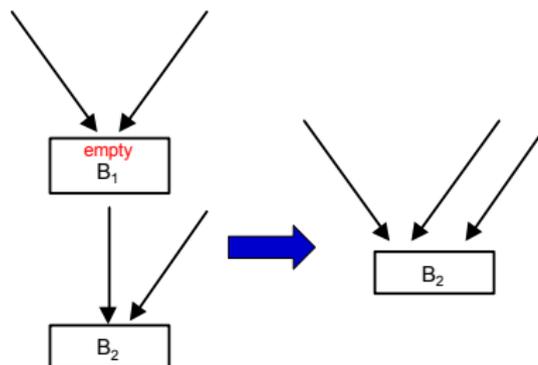
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

- Entsteht durch:
Gelöschte Operationen
in B1
- Voraussetzungen:
Leerer Block B1 endet
mit Sprung
- Vorgehen:
 - Verbiege eingehende
Kanten von B1 zu B2
 - Entferne B1



Quelle: C&T Fig. 10.4

OptComp

A. Koch

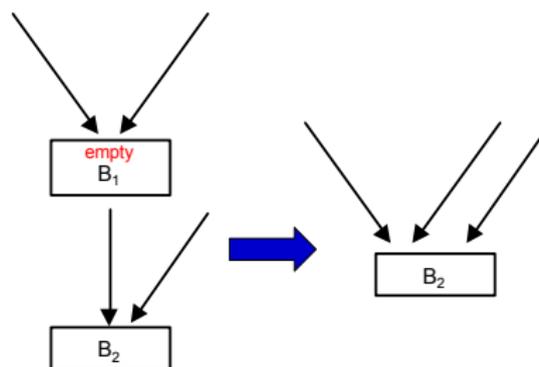
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

- Entsteht durch:
Gelöschte Operationen
in B1
- Voraussetzungen:
Leerer Block B1 endet
mit Sprung
- Vorgehen:
 - Verbiege eingehende
Kanten von B1 zu B2
 - Entferne B1



Quelle: C&T Fig. 10.4

OptComp

A. Koch

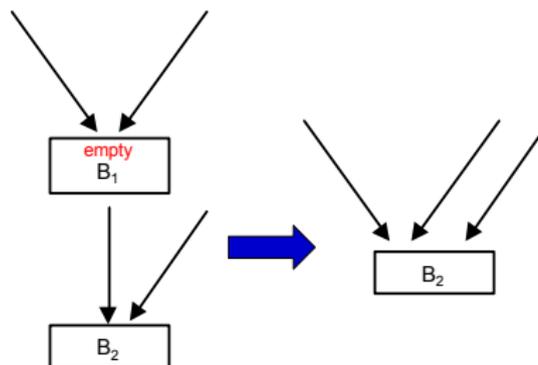
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung

- Entsteht durch:
Gelöschte Operationen
in B1
- Voraussetzungen:
Leerer Block B1 endet
mit Sprung
- Vorgehen:
 - Verbiege eingehende
Kanten von B1 zu B2
 - Entferne B1



Quelle: C&T Fig. 10.4

OptComp

A. Koch

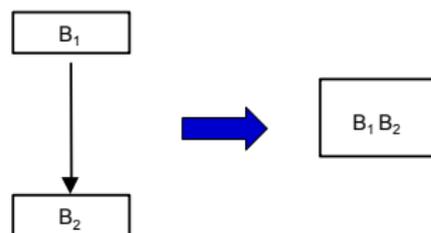
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassend

- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

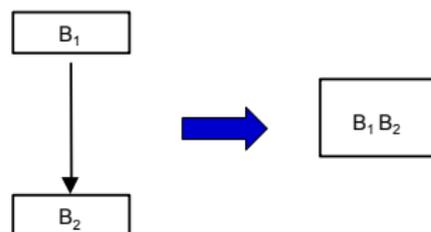
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

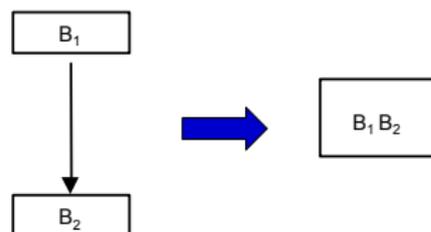
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

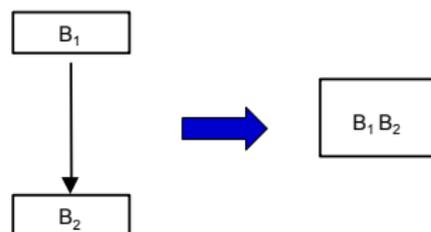
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



OptComp

A. Koch

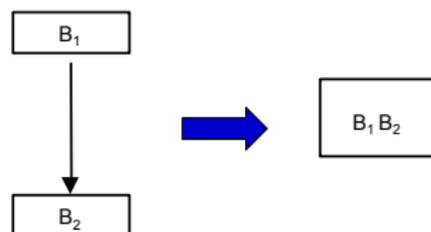
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

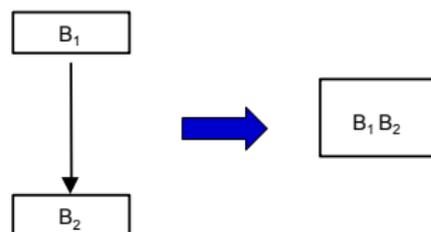
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

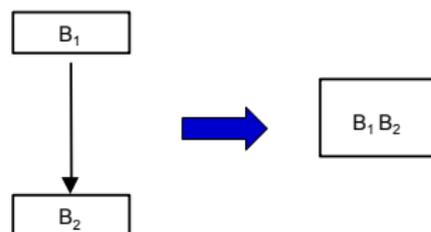
Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



Quelle: C&T Fig. 10.4

OptComp

A. Koch

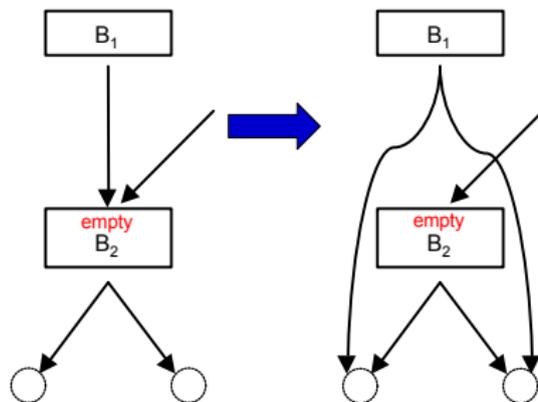
Organisatorisch

Dead Code
Elimination

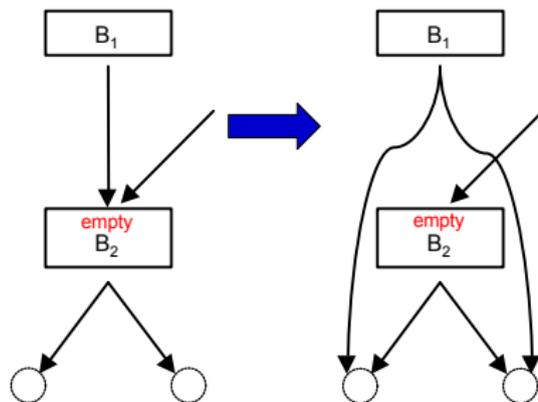
Bereinigen
des CFG

Zusammenfass

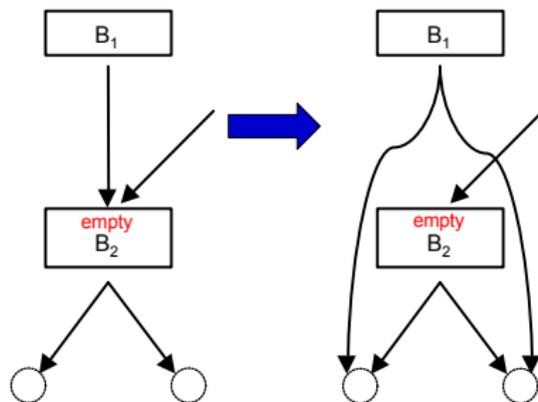
- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



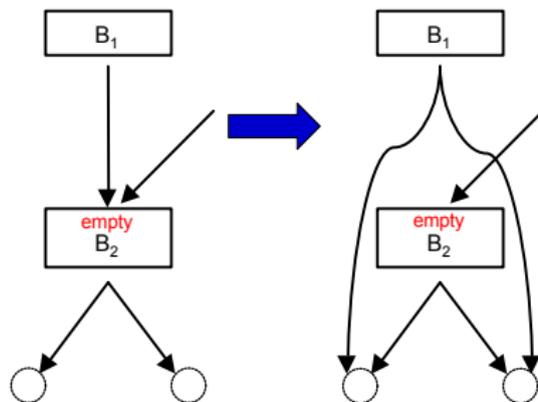
- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



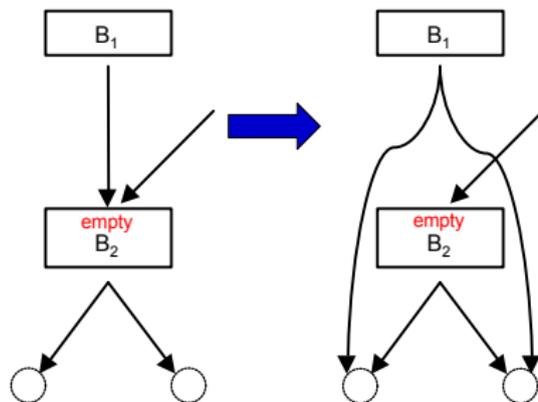
- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



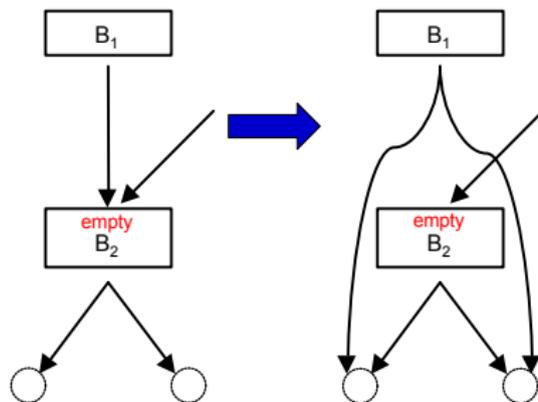
- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



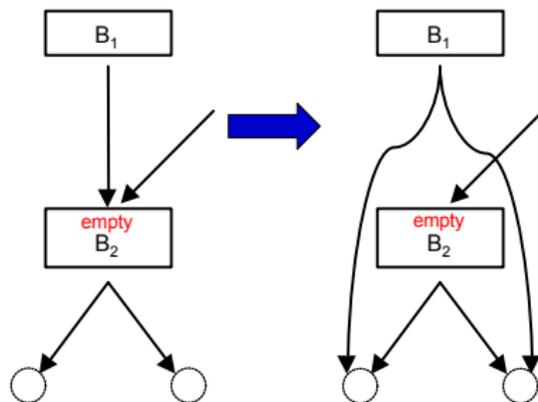
- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - Entferne redundante Verzweigungen
 - Entferne if mit true oder false Bedingung
 - Beseitige leere Blöcke
 - Entferne if mit true oder false Bedingung
 - Verschmelze Blöcke
 - Entferne if mit true oder false Bedingung
 - Ziehe Verzweigungen heraus
 - Entferne if mit true oder false Bedingung
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - Entferne redundante Verzweigungen
 - Beseitige redundante Blöcke
 - Beseitige leere Blöcke
 - Verschmelze Blöcke
 - Ziehe Verzweigungen heraus
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfass

- Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

```
CleanPass()
  foreach block i, in postorder
    if i ends in a branch then
      if both targets are identical then
        rewrite with a jump
    if i ends in a jump to j then
      if i is empty then
        merge i with j
      else if j has only one predecessor
        merge i with j
      else if j is empty & j has a branch then
        rewrite i's jump with j's branch

Clean()
  until CFG stops changing
  compute postorder
  CleanPass()
```

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassu

Zusammenfassung

- Erster Einblick in skalare Optimierung
- Modifikation des CFG
- Neue und alte Konzepte
 - Dominanz, Postdominanz, Dominanzgrenzen
- Dead Code Elimination: DEAD
- Bereinigen des CFG: CLEAN

OptComp

A. Koch

Organisatorisch

Dead Code
Elimination

Bereinigen
des CFG

Zusammenfassung