Compiler 2

8. Block: Registerallokation

TECHNISCHE UNIVERSITÄT DARMSTADT

Organisatorisches

TECHNISCHE UNIVERSITÄT DARMSTADT

- Klausuranfang nun 30 Minuten **früher**
  - Wegen unmittelbar anschliessender anderer Klausur
  - Nun 11:30-13:30 Uhr am 22.7.2013 in C205
- Keine Hilfsmittel erlaubt

Quellenangabe

TECHNISCHE UNIVERSITÄT DARMSTADT

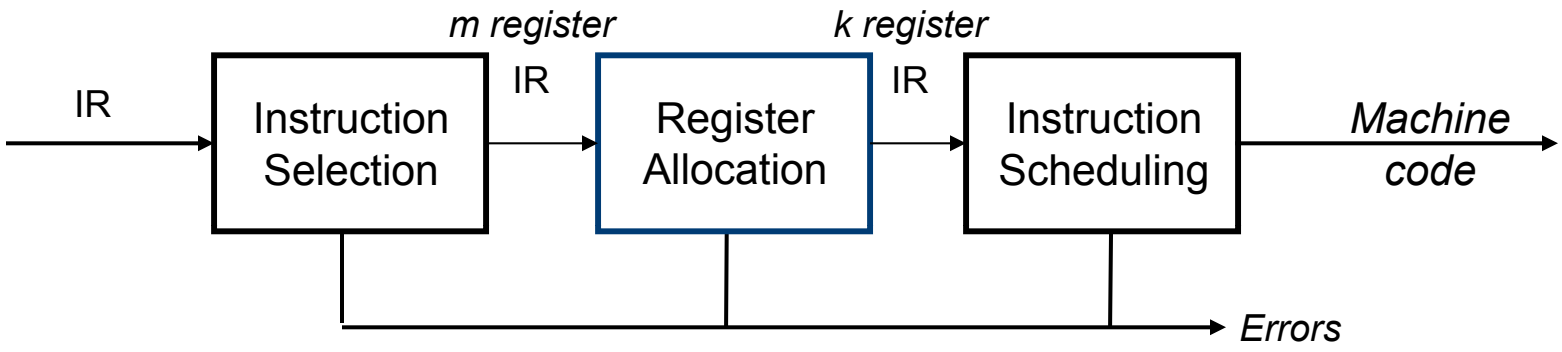
- Präsentation basiert auf Kapitel 13 von

  Engineering a Compiler, 2nd Edition

  Keith Cooper & Linda Torczon
  Morgan-Kaufman Publishers 2012

- Folienpräsentation enthält Abbildungen und Beispiele aus Begleitmaterial zum Buch

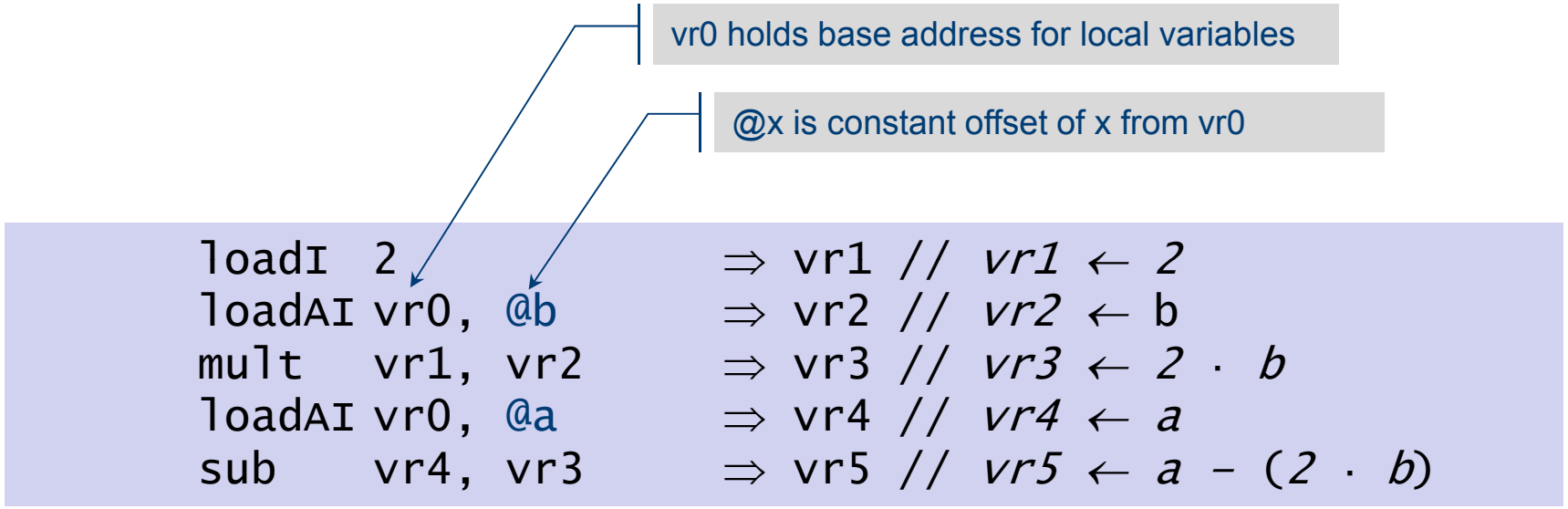
Einordnung in Compilefluß



- Teil von Compiler-Back End
- Erzeugt korrekten Code der nicht mehr als *k* **Maschinenregister** benötigt
- Vermeide Loads/Stores bei **Überbelegungen** (*spills*) des Registerfeldes
- Minimiere Platz zur Auslagerung von Werten aus überbelegten Registern
- **Laufzeiteffizienz** ist wichtig: In der Praxis maximal $O(n^2)$, nicht $O(2^n)$
  - Muss potentiell alle *n* Instruktionen bearbeiten

Beispiel

EAC2e verwendet ILOC, ähnlich Bantam TAC



- Virtuelle oder **Pseudo-Register** repräsentieren **Werte**
- Für jede Instruktion entscheiden
  - Welche Pseudo-register vr*X* in **echten** Maschinenregistern r*Y* halten?
  - **Einfach**: |Werte| <= |Maschinenregister|
  - **Komplizierter**: |Werte| > |Maschinenregister|

ILOC

TECHNISCHE UNIVERSITÄT DARMSTADT

- Intermediate Language for Optimizing Compilers
- Beschrieben in Anhang A von EAC2e

a - 2 x b			
	loadI	2	$\Rightarrow vr_1$
	loadAI	vr_0, @b	$\Rightarrow vr_2$
	add	vr_1, vr_2	$\Rightarrow vr_3$
	loadAI	vr_0, @a	$\Rightarrow vr_4$
	sub	vr_4, vr_3	$\Rightarrow vr_5$

- Semantik sehr ähnlich zu Bantam TAC
  - Unbegrenzt viele Pseudo-Register
- Operanden: I-Immediate, A-Basisadressregister, O-Offsetregister
  - Wenn nichts angegeben: Register

Aufgaben der Registerallokation

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wähle zu jedem Zeitpunkt die in Maschinenregistern gehaltenen Werte
- Erzeuge Instruktionen zum **Transfer von Werten** zwischen
  - Registern
  - Speicher

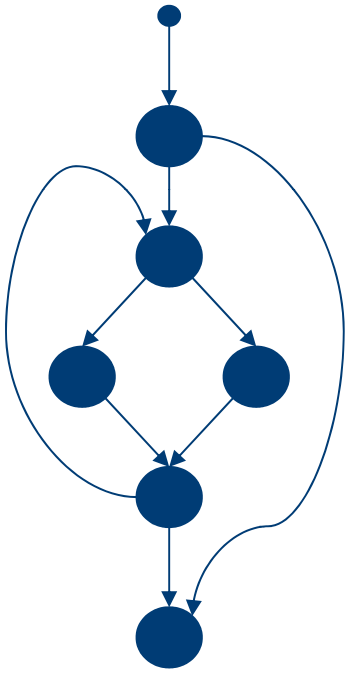
- **Minimiere Aufwand** für Transfer-Instruktionen
  - Dynamisch – Anzahl ausgeführter Instruktionen
  - Statisch – Anzahl abgespeicherter Instruktionen
  - Bei lokalem Vorgehen (Basisblöcke): Dynamisch ↔ Statisch

- Genauer
  - **Allokation** (*allocation*): Welche Werte sollen in Registern gehalten werden?
  - **Zuweisung** (*assignment*): In welchen Registern sollen Werte gehalten werden?
  - Compiler muss beide Aufgaben erledigen!

Lokale Registerallokation

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschränkung auf einzelne Basisblöcke
  - Liefert gute Ergebnisse innerhalb der Blöcke
  - Wird aber ineffizient an Blockgrenzen



Blocks in a Control-flow Graph (CFG)

Optimale Allokation ist hart

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Lokale Allokation**
  - In vereinfachten Fällen: O(n)
    - Werte haben nur eine Größe
    - Alle Werte müssen am Blockende in den Speicher geschrieben werden
    - Alle Speicherzugriffe haben die selben Kosten
  - Realistische Fälle: NP-vollständig
- **Lokale Zuweisung**
  - In vereinfachten Fällen: O(n)
    - Werte haben nur eine Größe
    - Keine Spills
  - Realistische Fälle: NP-vollständig
- **Globale Allokation**: NP-vollständig für $k \geq 1$ Register
- **Globale Zuweisung**: NP-vollständig

Effekt der SSA-Form

TECHNISCHE UNIVERSITÄT DARMSTADT

- Registerallokation auf SSA-Form geht an sich schneller
  - **Polynomiale Zeit**, mit niedrigem Grad des Polynoms
- Aber: Danach muss SSA-Form **aufgelöst** werden
  - Phi-Funktionen in Kopien auflösen
  - Benötigt wiederum Register
  - …
- Hier nicht diskutiert, erstmal grundsätzliches Vorgehen

ILOC und Registerallokation

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der Registerallokator braucht **keine** Kenntnisse über Bedeutung des Codes

- Lediglich **DEFs** und **USEs** sind relevant
  - DEFs können einen überbelegenden Wert in den Speicher schreiben
  - USEs können einen überbelegten Wert aus dem Speicher lesen

- DEFs und USEs leicht in ILOC-Instruktionen erkennbar

Beobachtung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Wert **lebt** (*is live*) zwischen seinen DEFs und USEs
  - Bestimme DEFs (x ← …) und USEs (… ← … x …)
- **Lebenszeit** eines Wertes (*live range*)
  - Anfang: Letzte DEF
  - Ende: Letzte USE
  - Darstellung innerhalb eines Basisblocks: Als Intervall [*i*,*j*]
    - *i*,*j* sind die fortlaufenden Nummern der Instruktionen in BB
- Lebenszeiten sind schwerer zu bestimmen bei **globalen** Verfahren

MAXLIVE und sein Einfluß

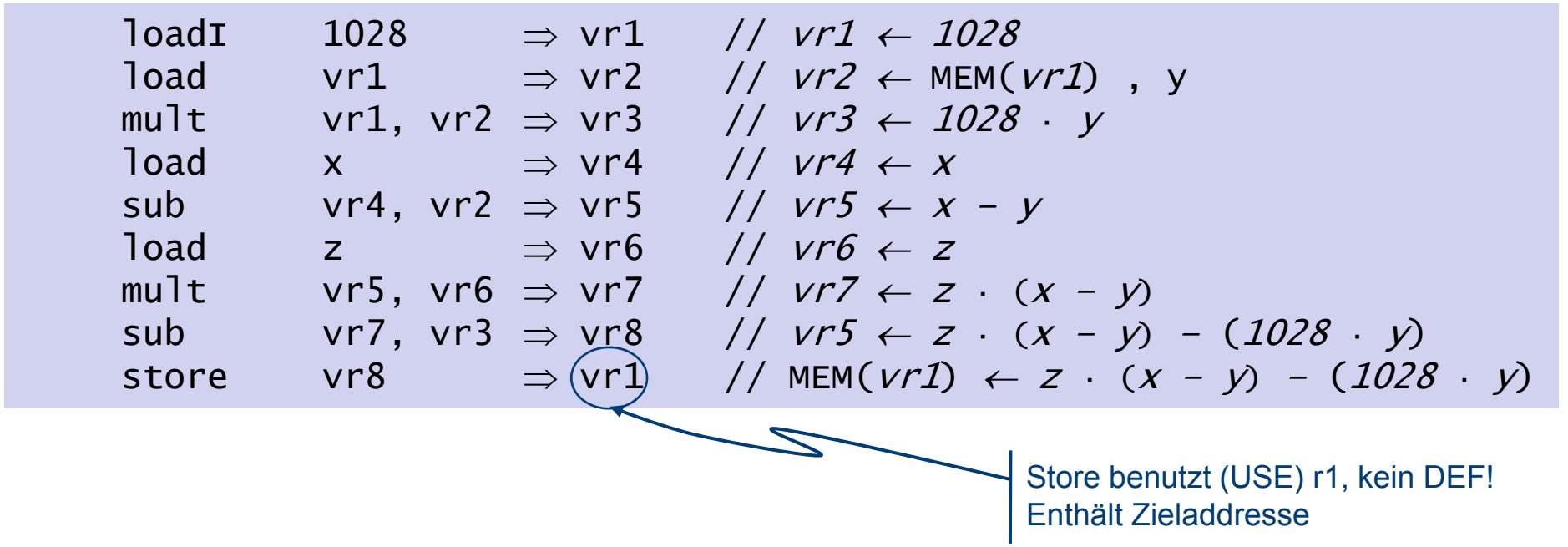
TECHNISCHE UNIVERSITÄT DARMSTADT

- **MAXLIVE**
  - Über alle Instruktionen *i* eines BBs …
    - …die Anzahl von Lebenszeit-Intervallen, in denen *i* enthalten ist.
  - Bedeutung: Anzahl der lebenden Werte (Pseudo-Register) zur Instruktion *i*

- **Interpretation**: Falls …
  - … MAXLIVE ≤ *k*: Allokation einfach
  - … MAXLIVE ≤ *k*: Keine Reserveregister zur Handhabung von Spills vorhalten
  - … MAXLIVE > *k*: einige Werte müssen im Speicher gehalten werden
  - … MAXLIVE > *k*: *F* Register reservieren, um Werte aus Speicher zurückzuholen
    - Beispiel: MIPS braucht mindestens *F*=1 Register für Speicherzugriff
      - Ausnahme bei MIPS?

Beispiel für MAXLIVE

Eingabecode



- Beispiel benutzt 1028 als
  - Adresse von *y*
  - Konstante in der Berechnung
  - Hier verwendet, um längere Lebenszeit zu konstruieren

Beispiel für MAXLIVE

Lebenszeiten


```
loadI   1028     ⇒ vr1   // vr1
load    vr1      ⇒ vr2   // vr1 vr2
mult    vr1,vr2  ⇒ vr3   // vr1 vr2 vr3
load    x        ⇒ vr4   // vr1 vr2 vr3 vr4
sub     vr4,vr2  ⇒ vr5   // vr1     vr3      vr5
load    z        ⇒ vr6   // vr1     vr3      vr5 vr6
mult    vr5,vr6  ⇒ vr7   // vr1     vr3               vr7
sub     vr7,vr3  ⇒ vr8   // vr1                           vr8
store   vr8      ⇒ vr1   //
```

Ein Pseudo-Register lebt nach einer Operation falls es vorher einen Wert bekommen hat (DEF), der in Zukunft benutzt wird (USE).

Beispiel für MAXLIVE

Bestimmung von MAXLIVE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
loadI   1028     ⇒ vr1   // vr1
load    vr1      ⇒ vr2   // vr1 vr2
mult    vr1,vr2  ⇒ vr3   // vr1 vr2 vr3
load    x        ⇒ vr4   // vr1 vr2 vr3 vr4
sub     vr4,vr2  ⇒ vr5   // vr1     vr3     vr5
load    z        ⇒ vr6   // vr1     vr3     vr5 vr6
mult    vr5,vr6  ⇒ vr7   // vr1     vr3             vr7
sub     vr7,vr3  ⇒ vr8   // vr1                         vr8
store   vr8      ⇒ vr1   //
```

MAXLIVE ist 4

Wichtig: Bei store ist RHS ein USE, kein DEF

Berechne Mengen lebender Variablen in Rückwärtsrichtung:

1. Beginn mit leerer Menge
2. In jeder Instruktion: entferne RHS, nehme Operanden aus LHS auf

Top-Down und Bottom-Up
Vorgehensweisen bei der Registerallokation

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Top-Down Allokator**
  - Basiert auf Angaben externer Funktion zur Bestimmung von "Wichtigkeit"
    - Üblich: Am häufigsten benutzte Werte sind wichtig
  - Weise dann Register an Werte in absteigender Wichtigkeit zu
  - Halte Reserveregister zurück, um Speicherzugriffe ausführen zu können
    - Ablegen/Wiederholen von überbelegten Werten

- **Bottom-Up Allokator**
  - Bestimme genaue DEF/USE-Struktur für jeden Eingabecode
  - Baue dann konstruktiv Gesamtlösung aus Teillösungen zu jedem Schritt auf
  - Behandelt alle Werte gleich (keine "Prioritäten" mehr)

TOP-DOWN ALLOKATOR

Grundlagen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Idee**
  - Halte am häufigsten benutzte Werte in Registern
  - Reserviere *F* Register zum Ablegen/Wiederholen von Werte in/aus Speicher
- **Algorithmus**
  - Bestimme Priorität von Werten: Häufiger auftretende haben höhere
  - Alloziiere die ersten k – F Werte an Register
    - Alle anderen verbleiben in Speicher und werden nur bei Bedarf geholt
  - Schreibe Code um
    - Einfügen von LOAD/STOREs für nur im Speicher abgelegte Werte
- **In den 70er/80er Jahren**
  - Manuelles Vorgehen: Schlüsselwort `register` in C
    - Weise Compiler manuell an, diese Variable in Register zu halten

Anzahlen von Registern

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Bestimme *F***: Wieviele reservierte Register werden benötigt?
  - Register benötigt für
    - Berechnung der Speicheradresse
    - Speichern des geladenen Wertes
  - Genaue Anzahl hängt von Zielarchitektur ab
    - Bei MIPS: 0…1 Register für Speicheradresse
    - Typische Instruktionen haben zwei Operanden, also Register für max. zwei Werte
      - Falls Speicheroperationen nicht auf beliebigen Registern arbeiten können
  - Reserviere diese Anzahl von Registern für Spilling

- **Sonderfall**: $k\text{-}F < |\text{Werte}| < k$
  - Einfachste Vorgehensweise: Genau auf Sonderfall testen

Beispiel für Top-Down Allokator

Ausgangssituation

TECHNISCHE UNIVERSITÄT DARMSTADT

```
loadI   1028    ⇒ vr1  // vr1
load    vr1     ⇒ vr2  // vr1 vr2
mult    vr1,vr2 ⇒ vr3  // vr1 vr2 vr3
load    x       ⇒ vr4  // vr1 vr2 vr3  vr4
sub     vr4,vr2 ⇒ vr5  // vr1      vr3      vr5
load    z       ⇒ vr6  // vr1      vr3      vr5 vr6
mult    vr5,vr6 ⇒ vr7  // vr1      vr3               vr7
sub     vr7,vr3 ⇒ vr8  // vr1                            vr8
store   vr8     ⇒ vr1  //
```

- **Annahmen**: k=3, F=2 (für zwei Operanden und ein Ergebnis)
  - Hier kein separates Register für Speicheradresse erforderlich (ähnl. MIPS $0)
  - Beliebige Register als Quelle/Ziel in Speicheroperation
- **Auftrittshäufigkeiten**
  - |vr1|=4, |vr2|=3, |vr3|=2, |vr4|=2, |vr5|=2, |vr6|=2, |vr7|=2, |vr8|=2

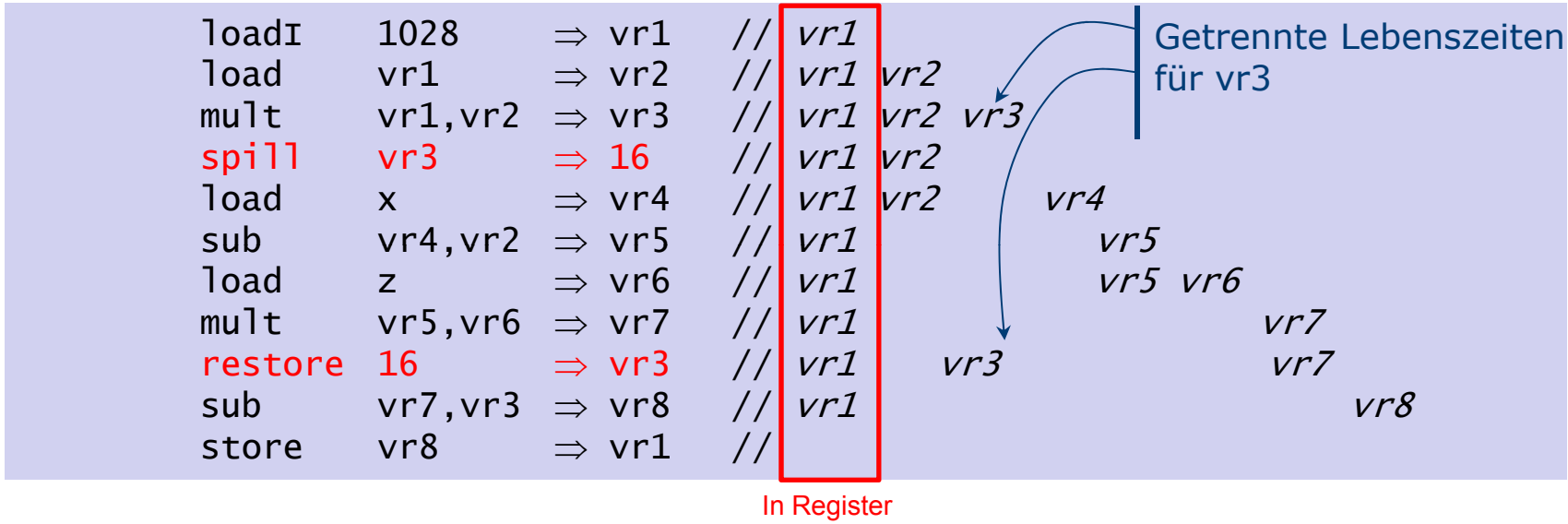
Beispiel für Top-Down Allokator

Allokation

- Es dürfen maximal k=3 Werte in einer Instruktion am Leben sein
- Wenn mehr: Verschiebe Wert mit niedrigster Priorität in Speicher
  - Auftreten: |vr1|=4, |vr2|=3, |vr3|=2, |vr4|=2, |vr5|=2, |vr6|=2, |vr7|=2, |vr8|=2
  - Priorität: vr1 > vr2 > vr3 = vr4 = vr5 = vr6 = vr7 = vr8

Beispiel Top-Down Allokator

Spill/Restore

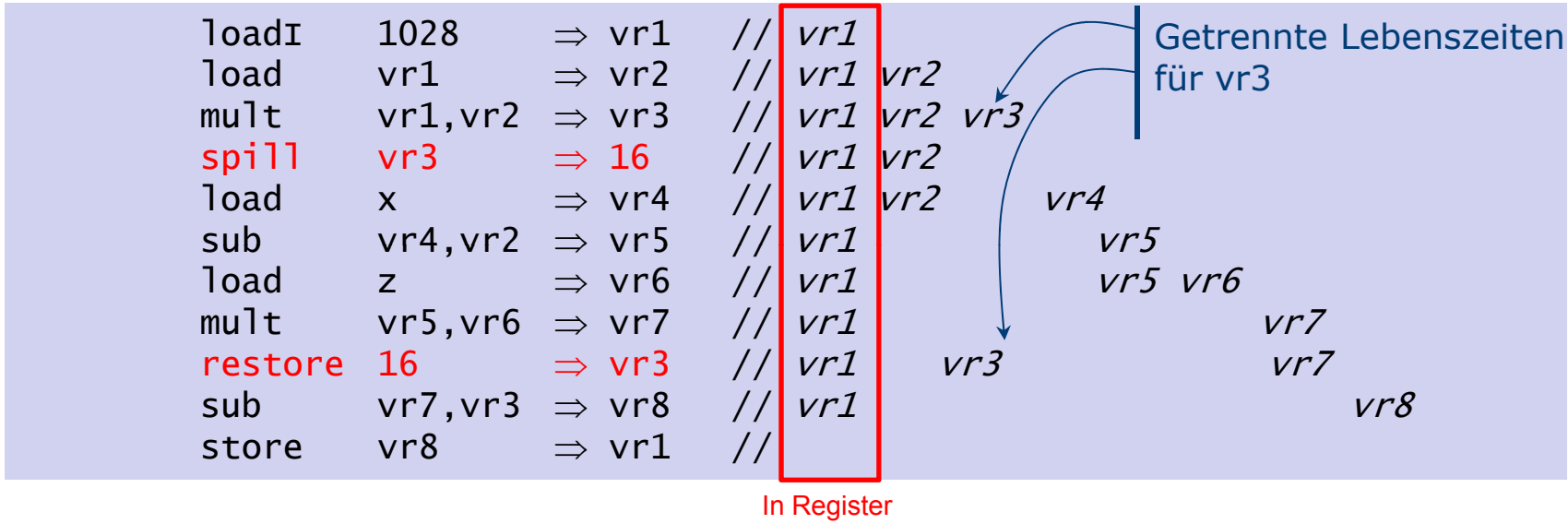


- **Spill/Restore Instruktionen** hier nur beispielhaft, bei MIPS benutzbar:
  - `sw $t3,0x10($0)`
  - `lw $t3,0x10($0)`

- vr3 hat nun **zwei** getrennte Lebenszeiten!

Beispiel Top-Down Allokator

Effekt der Aufteilung von Lebenszeiten



- **Kürzere Lebenszeiten** für vr3 überlappen sich mit weniger Lebenszeitintervallen anderer Werte
- Mit jeder Entscheidung für Spill wird Allokationsproblem **vereinfacht**
  - Siehe: iteratives Vorgehen
- Code ist nun **legal**: Maximal drei Pseudo-Register in Benutzung
- Braucht aber zwei weitere Instruktionen

Diskussion Top-Down Allocator

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktioniert
- Problem: Häufigkeit des Auftretens ist naives Maß für Wichtigkeit

- Gegenbeispiel
  - vr1 tritt extrem häufig am Anfang des BBs auf, danach nicht mehr
  - Trotzdem wird Maschinenregister den ganzen BB über für vr1 reserviert

- Lösungsansatz mit Bottom-Up Allokator

BOTTOM-UP ALLOKATOR

Grundlagen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Idee**
  - Konzentriert sich auf das Ersetzen von Werten in Registern anstatt auf Allokation von Registern
  - Halte Werte die "bald" wieder gebraucht werden in Registern
- **Algorithmus**
  - Beginne mit komplett unbelegten Registern
  - Nimmt an, dass alle Werte im Speicher stehen
  - Hole Werte nur bei Bedarf in Register
  - Wenn kein Register verfügbar, gebe eines frei
- **Ersetzen von Werten bei Freigabe**
  - Gebe Register mit Wert frei, der am weitesten in der Zukunft gebraucht wird
  - Bevorzuge unmodifizierte (*clean*) vor modifizierten (*dirty*) Werten
  - Ähnliche Strategie wie bei Paging in virtuellem Speicher

Beispiel für Bottom-Up Allokator

Gleicher Code wie vorher

TECHNISCHE UNIVERSITÄT DARMSTADT

```
loadI   1028    ⇒ vr1  // vr1
load    vr1     ⇒ vr2  // vr1 vr2
mult    vr1,vr2 ⇒ vr3  // vr1 vr2 vr3
load    x       ⇒ vr4  // vr1 vr2 vr3  vr4
sub     vr4,vr2 ⇒ vr5  // vr1      vr3      vr5
load    z       ⇒ vr6  // vr1      vr3      vr5 vr6
mult    vr5,vr6 ⇒ vr7  // vr1      vr3               vr7
sub     vr7,vr3 ⇒ vr8  // vr1                            vr8
store   vr8     ⇒ vr1  //
```

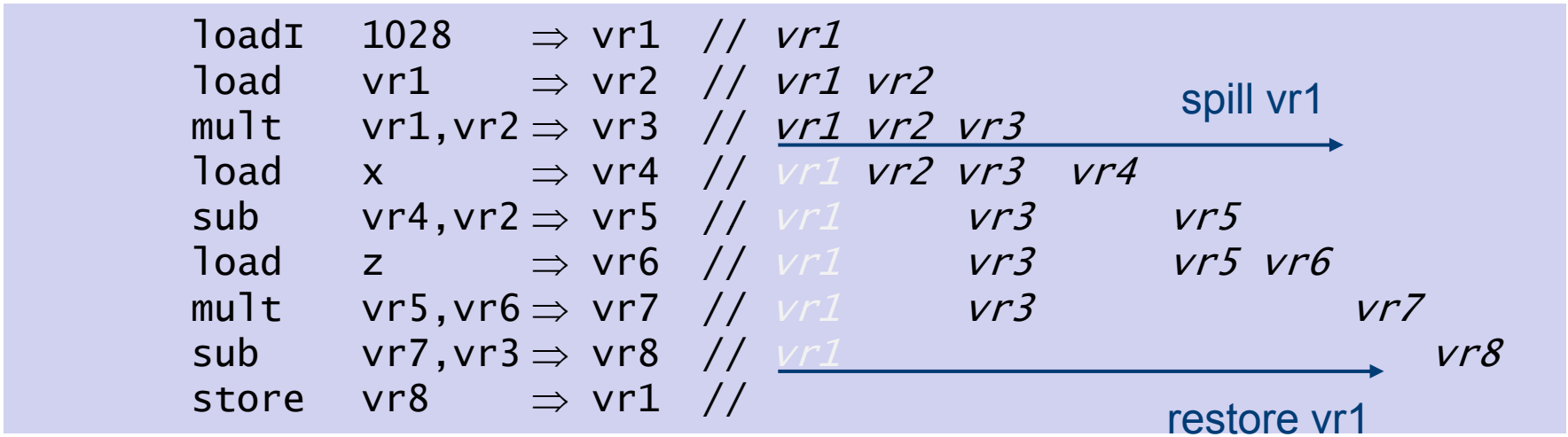
Ab hier alle Register Belegt.

- k=3, F=0

Embedded Systems & Applications

Beispiel für Bottom-Up Allokator

Nun Spilling des Wertes mit entfernter Benutzung



Beispiel für Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
loadI   1028     ⇒ vr1   // vr1
load    vr1      ⇒ vr2   // vr1 vr2
mult    vr1,vr2  ⇒ vr3   // vr1 vr2 vr3
spill   vr1      ⇒ 20    //     vr2 vr3
load    x        ⇒ vr4   //     vr2 vr3 vr4
sub     vr4,vr2  ⇒ vr5   //         vr3      vr5
load    z        ⇒ vr6   //         vr3      vr5 vr6
mult    vr5,vr6  ⇒ vr7   //         vr3               vr7
sub     vr7,vr3  ⇒ vr8   //                                vr8
restore 20       ⇒ vr1   // vr1                            vr8
store   vr8      ⇒ vr1   //
```

- Zu jeder Instruktion maximal drei Werte am Leben
- vr1 hat nun zwei getrennte Lebenszeitintervalle
  - Überlappen sich mit weniger anderen Lebenszeiten
  - Vereinfachen Allokationsproblem

Datenstrukturen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Eine Klasse von Maschinenregistern
  - Size: Anzahl von Registern in der Klasse
  - Für jedes Register in der Klasse
    - Namen des virtuellen Registers in diesem Maschinenregister
    - Distanz bis zum nächsten USE des virtuellen Registers
    - Marker, ob dieses Maschinenregister gerade verwendet wird
  - Einen Stack verfügbarer Maschinenregister mit Stapelzeiger

```
initialize(class,size) {
 class.Size ← size;

 for i ← size-1 to 0 do
  class.Name[i] ← -1;
  class.Next[i] ← ∞;
  class.Free[i] ← true;
  push(i,class);

 class.StackTop = size-1;
}
```

Quelle der Erklärung und des Beispiels: Pedro Diniz, USC CSCI 565, S 2011

Funktionen 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- **`class(vr`$_x$`)`** gibt Klasse von Maschinenregistern zurück, die vrx aufnehmen können
- **`ensure(vr`$_x$`,class`**) bestimmt schon vergebenes Register für vrx in der Klasse
  - falls noch keines vergeben: vergebe ein neues und lade den Inhalt aus Speicher
- **`allocate(vr`$_x$`,class`**) alloziert ein neues Register aus der Klasse
  - gibt bevorzugt ein freies zurück
  - falls alles belegt: das Register mit am weitesten in der Zukunft liegenden USE
    - sichert vorher dessen Inhalt in den Speicher

Funktionen 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- **`free(vr`$_x$`,class`**) gibt ein benutztes Register der Klasse frei
  - reinitialisiert seine Attribute
  - legt es wieder auf den Stack freier Register der Klasse

- **`dist(vr`$_x$**) Distanz zur Instruktion mit nächsten USE von vrx

ensure(vr_x,class)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
reg ensure(vr,class){
  r ← find(vr,class);   // Schon Maschinenregister alloziiert?
  if (r exists) then
    result ← r;
  else
    result ← allocate(vr,class);
    emit code to move vr into r;   // hole Wert aus Speicher
  end
  return result;
}
```

allocate(vr$_x$,class)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
reg allocate(vr,class) {
  if (class.StackTop ≥ 0) then // ein Maschinenregister frei?
    r ← pop(class);
  else
    r ← findMaxNext(class); // r mit am weitesten entfernten USE
    if (r is dirty) then
     emit code to save r in memory at address for vr;
  end

  // vergebe Maschinenregister neu
  class.Name[r] ← vr;
  class.Next[r] ← -1;       // Temp. Marker: wird aktuell benutzt,
                            // nicht sofort wieder freigeben!
  class.Free[r] ← false;

  return r;
}
```

free(vr$_x$,class)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
void free(i,class) {
  if (class.Free[i]) then        // falls nicht ohnehin frei
    push(i,class);               // lege auf Freistack
    class.Name[i] ← -1;          // und reinitialisiere
    class.Next[i] ← ∞;
    class.Free[i] ← true;
 end
}
```

Hauptfunktion

Eingabe : Basisblock B
Ausgabe: B mit umgeschriebenen Instruktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
foreach instr i: "vr_i3 ← vr_i1 op vr_i2" ∈ B
{

  rx ← ensure(vr_i1,class(vr_i1));
  ry ← ensure(vr_i2,class(vr_i2));

  if(vr_i1 is not needed after i)then
    free(vr_i1,class(vr_i1));
  if(vr_i2 is not needed after i)then
    free(vr_i2,class(vr_i2));

  rz ← allocate(vr_i3,class(vr_i3));
  rewrite i as "rz ← rx op ry"

  if (vr_i1 is needed after i) then
    class.Next[rx] = dist(vr_i1);

  if (vr_i2 is needed after i) then
    class.Next[ry] = dist(vr_i2);

  class.Next[rz] = dist(vr_i3);
}
```

Diskussion Algorithmus 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Berücksichtige **erst** Operanden vr_{i1} und vr_{i2}
  - … falls nicht mehr gebraucht, freigeben
  - Kann möglicherweise gleiches Register verwenden: … ← **`vr`$_b$ `op vr`$_b$**
- **Anschließend** Ergebnis vr_{i3} behandeln
  - Kann evtl. freigebene Operandenregister sofort wiederverwenden
- **`allocate`** setzt **`Next`** temporär auf -1, um sofortige Neubenutzung zu unterbinden
  - Korrekter Wert wird erst später eingetragen
    - **`dist(vrx)`** bei Wiederbenutzung

Diskussion Algorithmus 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Intuitive Vorgehensweise**
  - Nehme an, dass alle Maschinenregister frei sind
  - Vergebe Maschinenregister, solange freie verfügbar sind
  - Falls keine verfügbar, lagere Wert in Speicher aus
    - Den am weitesten in der Zukunft liegenden
  - Und verwende dessen Maschinenregister erneut

- **Rechtfertigung**
  - Führe so viele nützliche Instruktionen aus, wie möglich
  - … bevor ein Wert wieder geladen werden muss

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$$vr_3 \leftarrow vr_1 \ op \ vr_2$$
$$vr_5 \leftarrow vr_4 \ op \ vr_1$$
$$vr_6 \leftarrow vr_5 \ op \ vr_6$$
$$vr_7 \leftarrow vr_3 \ op \ vr_2$$

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$vr_3 \leftarrow vr_1 \text{ op } vr_2$
$vr_5 \leftarrow vr_4 \text{ op } vr_1$
$vr_6 \leftarrow vr_5 \text{ op } vr_6$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	-1	-1	-1
Next	∞	∞	∞
Free	T	T	T

Stack

2
1
0

Top = 2

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$vr_3 \leftarrow r_0 \ op \ vr_2$
$vr_5 \leftarrow vr_4 \ op \ vr_1$
$vr_6 \leftarrow vr_5 \ op \ vr_6$
$vr_7 \leftarrow vr_3 \ op \ vr_2$

Size = 3

	0	1	2
Name	vr_1	-1	-1
Next	1	∞	∞
Free	F	T	T

Stack

2
1

Top = 1

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$vr_3 \leftarrow r_0 \text{ op } r_1$
$vr_5 \leftarrow vr_4 \text{ op } vr_1$
$vr_6 \leftarrow vr_5 \text{ op } vr_6$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	vr_1	vr_2	-1
Next	1	3	∞
Free	F	F	T
Stack	2		

Top = 0

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$vr_5 \leftarrow vr_4 \text{ op } vr_1$
$vr_6 \leftarrow vr_5 \text{ op } vr_6$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	vr_1	vr_2	vr_3
Next	1	3	3
Free	F	F	F

Stack Top = -1

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0$ op r_1
$vr_5 \leftarrow \underline{vr_4}$ op vr_1
$vr_6 \leftarrow vr_5$ op vr_6
$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_1	vr_2	vr_3
Next	1	3	3
Free	F	F	F

Stack Top = -1

Problem: Kein Register frei für vr_4, **allocate** sucht nach Wert mit am weitesten entferntem USE für Spill. Hier bspw. vr_2

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$r_1 \rightarrow \text{mem}$
$vr_5 \leftarrow r_1 \text{ op } vr_1$
$vr_6 \leftarrow vr_5 \text{ op } vr_6$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	vr_1	vr_4	vr_3
Next	1	-1	3
Free	F	F	F

Stack Top = -1

Spill von vr_2
verwende r_1 wieder für vr_4

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0$ op r_1
$r_1 \rightarrow$ mem
$vr_5 \leftarrow r_1$ op vr_1
$vr_6 \leftarrow vr_5$ op vr_6
$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_1	-1	vr_3
Next	1	∞	3
Free	F	T	F
Stack	1		

Top = 0

vr_4 wird nicht mehr gebraucht, r_1 unmittelbar nach **ensure** freigeben

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$r_1 \rightarrow \text{mem}$
$vr_5 \leftarrow r_1 \text{ op } r_0$
$vr_6 \leftarrow vr_5 \text{ op } vr_6$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	-1	-1	vr_3
Next	∞	∞	3
Free	T	T	F

Stack

1
0

Top = 1

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$r_1 \rightarrow \text{mem}$
$r_0 \leftarrow r_1 \text{ op } r_0$
$vr_6 \leftarrow vr_5 \text{ op } vr_6$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	vr_5	-1	vr_3
Next	1	∞	3
Free	F	T	F
Stack	1		

Top = 0

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$r_1 \rightarrow \text{mem}$
$r_0 \leftarrow r_1 \text{ op } r_0$
$vr_6 \leftarrow r_0 \text{ op } vr_6$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	vr_5	-1	vr_3
Next	-1	∞	3
Free	F	T	F
Stack	1		

Top = 0

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$r_1 \rightarrow \text{mem}$
$r_0 \leftarrow r_1 \text{ op } r_0$
$vr_6 \leftarrow r_0 \text{ op } r_1$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	vr_5	vr_6	vr_3
Next	-1	-1	3
Free	F	F	F

Stack Top = -1

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$r_1 \rightarrow \text{mem}$
$r_0 \leftarrow r_1 \text{ op } r_0$
$r_1 \leftarrow r_0 \text{ op } r_1$
$vr_7 \leftarrow vr_3 \text{ op } vr_2$

Size = 3

	0	1	2
Name	vr_5	vr_6	vr_3
Next	-1	-1	3
Free	F	F	F

Stack Top = -1

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0 \text{ op } r_1$
$r_1 \rightarrow \text{mem}$
$r_0 \leftarrow r_1 \text{ op } r_0$
$r_1 \leftarrow r_0 \text{ op } r_1$
$vr_7 \leftarrow r_2 \text{ op } vr_2$

Size = 3

	0	1	2
Name	-1	-1	vr_3
Next	∞	∞	-1
Free	T	T	F

Stack

0
1

Top = 1

Embedded Systems & Applications

Beispiel Bottom-Up Allokator

$r_2 \leftarrow r_0$ op r_1
$r_1 \rightarrow$ mem
$r_0 \leftarrow r_1$ op r_0
$r_1 \leftarrow r_0$ op r_1
$r_1 \leftarrow$ mem
$vr_7 \leftarrow r_2$ op r_1

Size = 3

	0	1	2
Name	-1	vr_2	vr_3
Next	∞	-1	-1
Free	T	F	F

Stack: 0

Top = 0

Vorher gespilltes vr_2 zurückholen

Beispiel Bottom-Up Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

$r_2 \leftarrow r_0$ op r_1
$r_1 \rightarrow$ mem
$r_0 \leftarrow r_1$ op r_0
$r_1 \leftarrow r_0$ op r_1
$r_1 \leftarrow$ mem
$r_0 \leftarrow r_2$ op r_1

Size = 3

	0	1	2
Name	vr_7	vr_2	vr_3
Next	-1	-1	-1
Free	F	F	F

Stack Top = -1

Vorher gespilltes vr_2 zurückholen

Hier nicht gezeigt:

Laden der initialen Werte der Register aus Speicher
Zurückschreiben der Ergebnisse in Speicher

Embedded Systems & Applications

Diskussion: Dirty / Clean

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schreiben von r1 in Speicher **nicht nötig**
- r1 wurde im Code nicht modifiziert
  - Ist *Clean*

- **Idee**
  - Verwende bevorzugt Register wieder, die Clean sind
  - Wert muss nicht zurückgeschrieben werden

- **Kann aber ineffizient sein**
  - Wenn ein Dirty-Wert sehr weit entfernt benutzt wird
  - Rückschreiben in Speicher wäre besser, …
  - … als Dirty Wert sehr lange im Register zu halten

```
r2  ← r0  op r1
r1  → mem //unnötig
r0  ← r1  op r0
r1  ← r0  op r1
r1  ← mem
r0 ← r2 op r1
```

Anforderungen an guten Allokator

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lebenszeiten von Werten bestimmen
- Überlappungen (*interference*) zwischen Lebenszeiten erkennen
- Kosten eines Register-Spills bestimmen (*Clean/Dirty*)
- Entscheidung, welche Lebenszeiten von Werten in Registern gehalten werden (*allocation*)
- Lebenszeiten aufteilen (*spilling, splitting*)
- Zuweisung von Maschinenregistern an Werte (*assignment*)
- Code-Generierung
  - Umschreiben auf Maschinenregister
  - Code zur Handhabung von Spills einfügen
  - (Code zum Rückschreiben von Ergebnissen einfügen)

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lokale Methoden sind besser als nichts
  - Ohne Registerallokator: Variablen nur im Speicher halten
  - Z.B. in der Basis-Version von Bantam
- Müssen in der Praxis aber auch schon NP-harte Probleme lösen
- Viele Vorgehensweisen möglich
- Bottom-Up besser als Top-Down
  - Bottom-Up berücksichtigt tatsächliche Programmstruktur
  - Top-Down verlässt sich auf die aggregierten Wichtigkeitsdaten

GLOBALE REGISTERALLOKATION

Übersicht

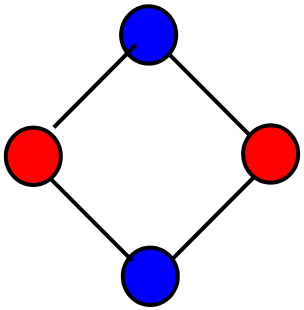
TECHNISCHE UNIVERSITÄT DARMSTADT

- Gängige Vorgehensweise für globale Registerallokation
  - Baue Konflikt- oder Interferenzgraph für CFG auf
  - Löse dann Graphenfärbungsproblem mit *k* Farben
    - Falls nicht möglich: Transformiere Code, so dass *k*-Einfärbung möglich ist
    - Jede Farbe entspricht dann einem Maschinenregister

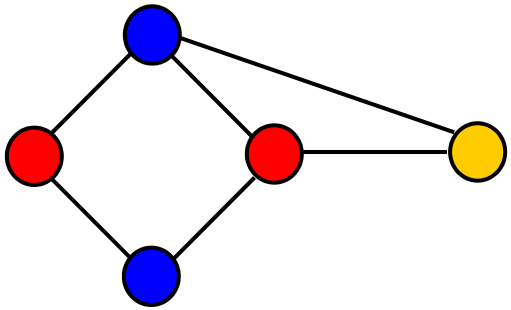
Exkurs: Graphenfärbungsproblem

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine durch Kante verbundenen Knoten haben dieselbe Farbe
- Für Einfärbung des kompletten Graphen werden maximal *k* Farben benötigt



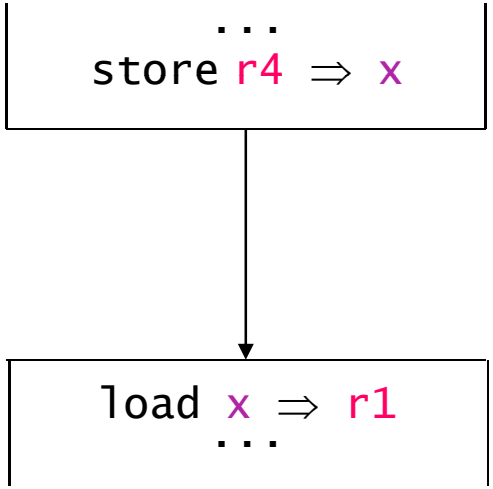
2-einfärbbar



3-einfärbbar

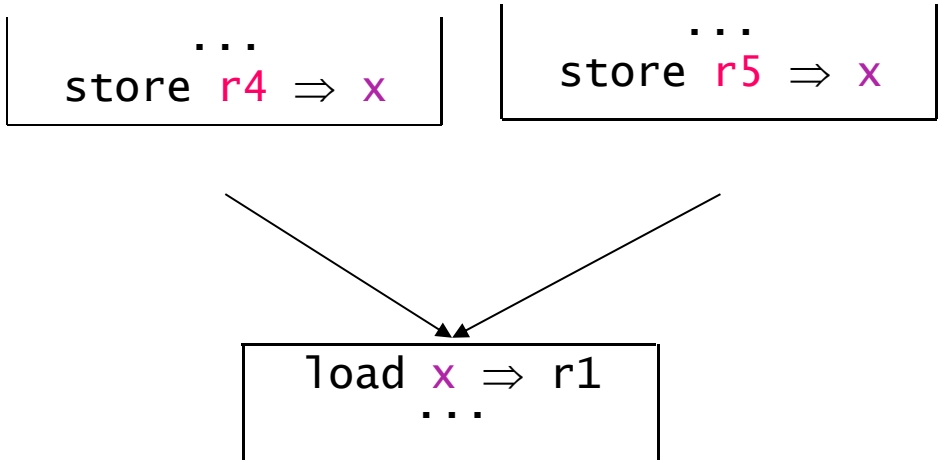
Schwierigkeiten 1

- Triviales Vorgehen
  - Alle Werte im Speicher zu Blockanfang
  - Dann lokale Registerallokation
  - Alle Werte zum Blockende in Speicher schreiben
- Ineffizient: Im Beispiel
  - Könnte Load durch Move ersetzen
  - Idealerweise ganz weglassen
  - Stattdessen gleiches Register verwenden
- Sinnvolle **Registerzuweisung** über Blockgrenzen hinweg



Schwierigkeiten 2

- Komplexer: Mehrere Vorgänger im CFG
- Registerzuweisung muß nun über drei Blöcke abgestimmt werden
- Spezialität: Schleife
  - Block ist sein eigener Vorgänger
- Macht Erweitern der lokalen Verfahren extrem kompliziert
  - … und potentiell langsam



Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grenzen zwischen Intra-Block und Inter-Block Bearbeitung aufgeben
- Alles mit globalem Verfahren rechnen

- **Registerallokation auf Basis von Graphenfärbung**

1 Baue Interferenzgraph G_I für Prozedur auf
   - Lebenszeiten sind schwerer zu bestimmen
   - G_I ist kein Intervallgraph
2 Berechne Einfärbung mit maximal *k* Farben
   - Minimale Einfärbung von allgemeinen Graphen ist NP-Vollständig
     - Heuristik benutzen
   - Platzierung von Spills nun sehr kritisch (Schleifen!)
3 Ordne Farben an Maschinenregister zu

Interferenzgraphen 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Interferenz**
  - Zwei Werte *x* und *y* interferieren falls eine Operation existiert, während der beide Live sind
  - Falls *x* und *y* interferieren, können sie nicht im gleichen Register abgelegt sein
- Offensichtlich erforderlich: Bestimmung der **Liveness** von Werten
- **Interferenzgraph** $G_I = (N_I, E_I)$
  - Knoten in N_I repräsentieren Lebenszeiten von Werten
  - Kanten in E_I repräsentieren einzelne Interferenzen
    - Für x, y $\in N_I$: (x,y) $\in E_I$ genau dann, wenn x und y interferieren
- *k*-Einfärbung von G_I kann betrachtet werden als Allokation auf *k* Register

Interferenzgraphen 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Lebenszeiten** bestimmen
  - Auf SSA-Form der Prozedur
    - Nun Low-Level IR bzw. Maschineninstruktionen mit virtuellen Registern
  - Bilde Vereinigungsmenge der Lebenszeiten der Argumente an Phi-Funktionen
  - Arbeite nun auf Lebenszeiten, nicht mehr auf virtuellen Registern

- Berechne **LIVE-Mengen** über Lebenszeiten
  - Mit iterativem Datenflusslöser
  - Übliche Gleichungen wie bei Live-Variablen, nun mit Lebenszeiten

- Iteriere über jeden Block, rückwärts über dessen Operationen
  - Konstruiere LIVENOW-Menge (nun per Operation), beginnend mit LIVEOUT(B)
  - Füge entsprechende Kanten in Graph hinzu
    - Vom Ergebnis der Operation zu allen Lebenszeiten in LIVENOW
    - Entferne Ergebnis aus LIVENOW
    - Füge Operanden zu LIVENOW hinzu

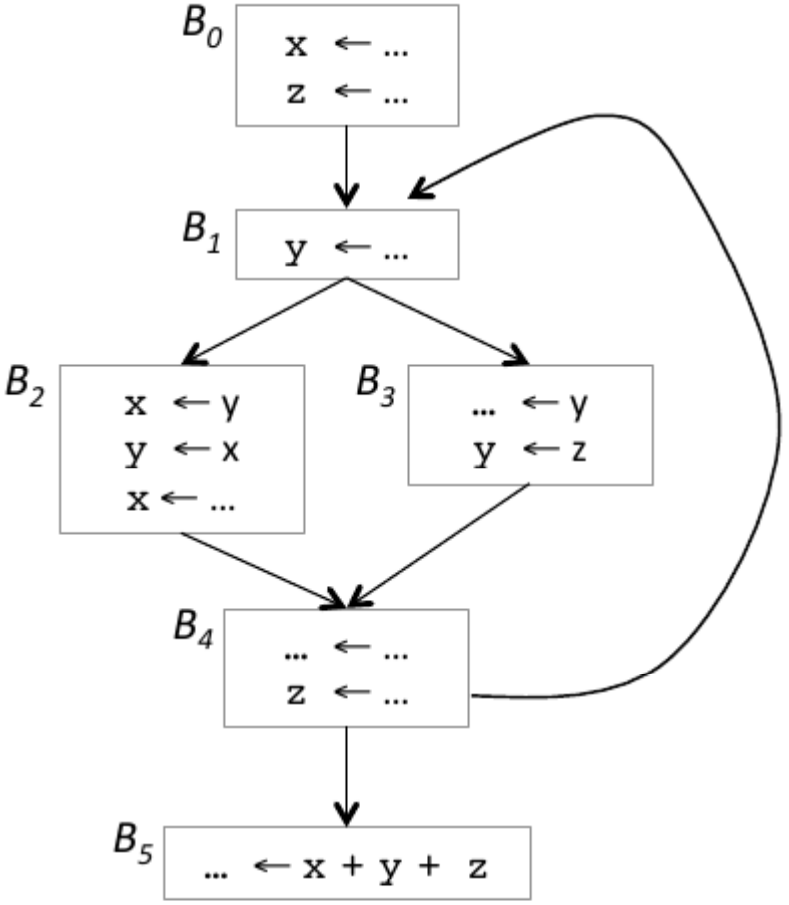
Lebenszeiten in CFGs 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Eine Lebenszeit L in einem CFG enthält DEFs und USEs von Werten
- Für jeden USE *u* in L müssen alle DEFs, die *u* erreichen auch in L enthalten sein
- Für jeden DEF *d* in L müssen alle USEs enthalten sein, die von *d* erreicht werden

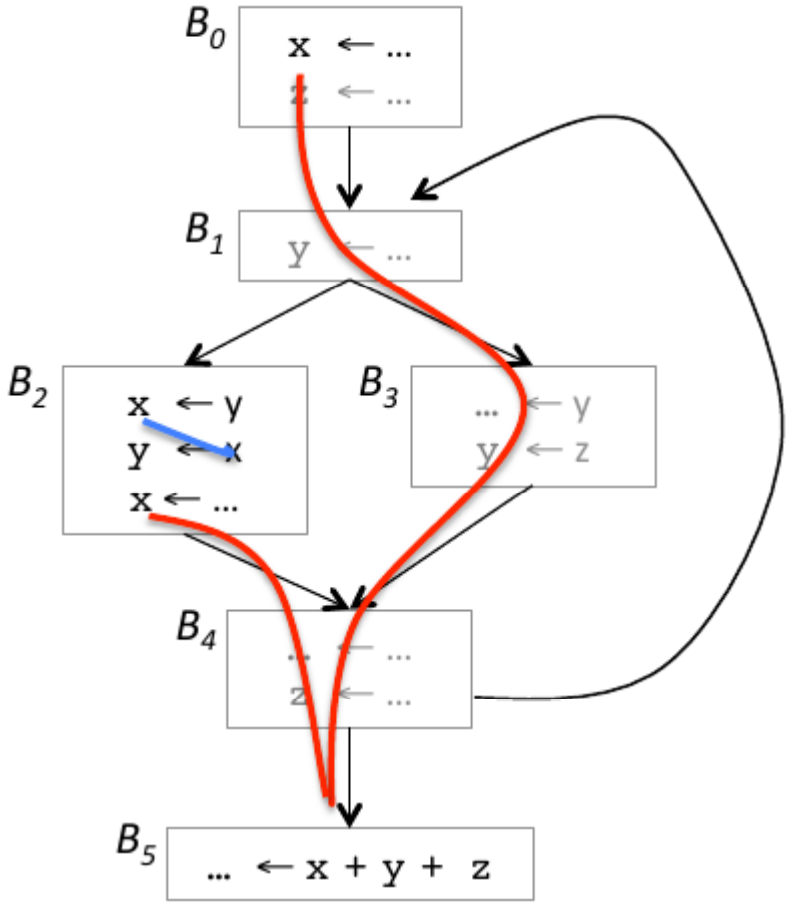
Lebenszeiten in CFGs 2

- Komplizierter als im lokalen Falle
- Beispiel: Lebenszeiten von *x*, *y* und *z*



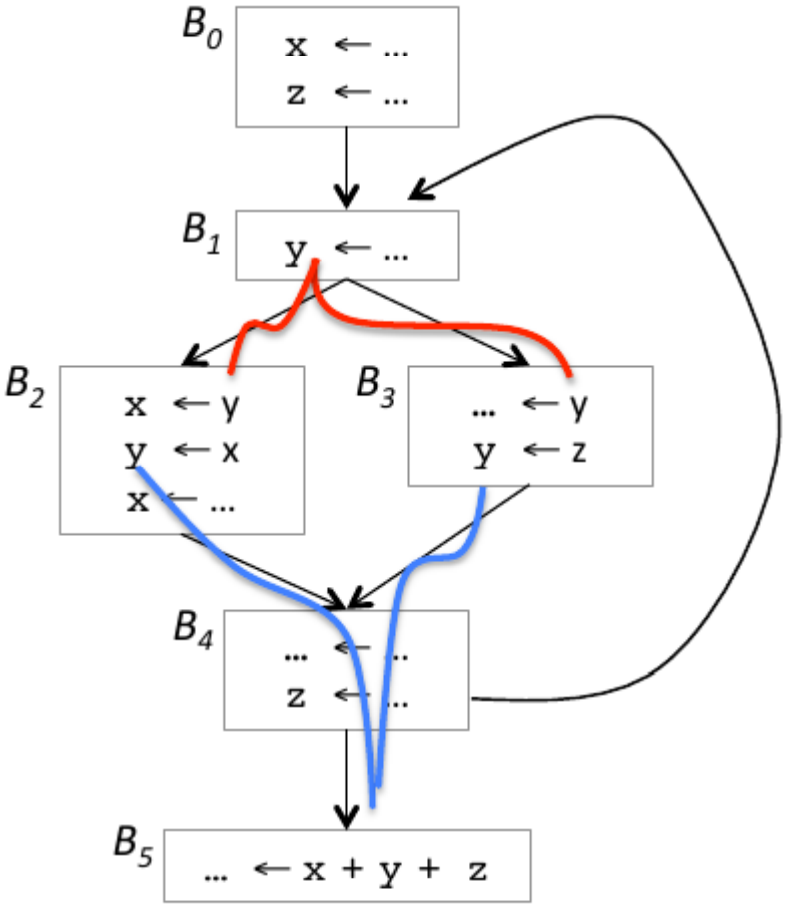
Lebenszeiten in CFGs 3

- Komplizierter als im lokalen Falle
- Beispiel: Lebenszeiten von *x*, *y* und *z*
  - *x* hat zwei getrennte Lebenszeiten



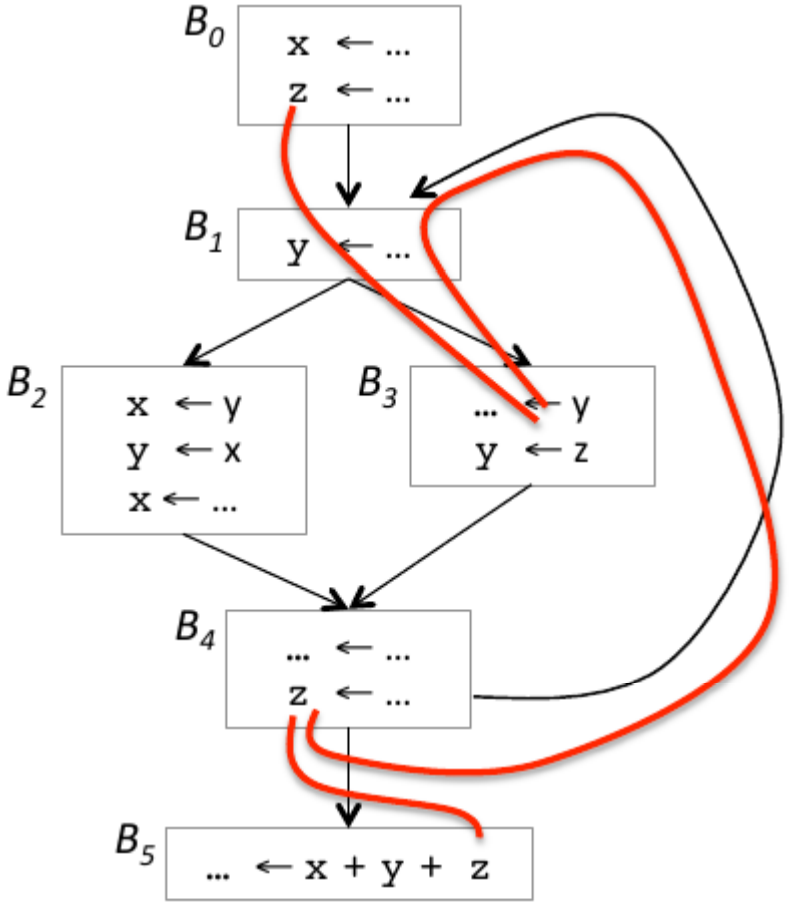
Lebenszeiten in CFGs 4

- Komplizierter als im lokalen Fall
- Beispiel: Lebenszeiten von *x*, *y* und *z*
  - *x* hat zwei getrennte Lebenszeiten
  - *y* hat zwei getrennte Lebenszeiten



Lebenszeiten in CFGs 5

- Komplizierter als im lokalen Fall
- Beispiel: Lebenszeiten von *x*, *y* und *z*
  - *x* hat zwei getrennte Lebenszeiten
  - *y* hat zwei getrennte Lebenszeiten
  - *z* hat nur eine Lebenszeit
    - *z* lebt niemals in B_2
- Bestimmung kann mühsam werden



Lebenszeiten in SSA-Form 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Deutlich einfacher in SSA-Form
  - Jeder Wert hat genau eine Definition
  - An Phi-Funktionen fliessen Werte zusammen

- Algorithmus
  - Wandele CFG ggf. in SSA-Form um (nur für Lebenszeitbestimmung!)
  - Betrachte zunächst jeden einzelnen SSA-Namen als einzelne Lebenszeit
  - Bei Phi-Funktion: Vereinige Lebenszeiten
    - der Parameter
    - des Ergebnisses
  - Verbliebene Mengen sind die Lebenszeiten der Werte
  - Benenne nun Werte im SSA-Graphen in Lebenszeiten um
    - Jede Lebenszeit entspricht einem virtuellen Register

Lebenszeiten in SSA-Form 2

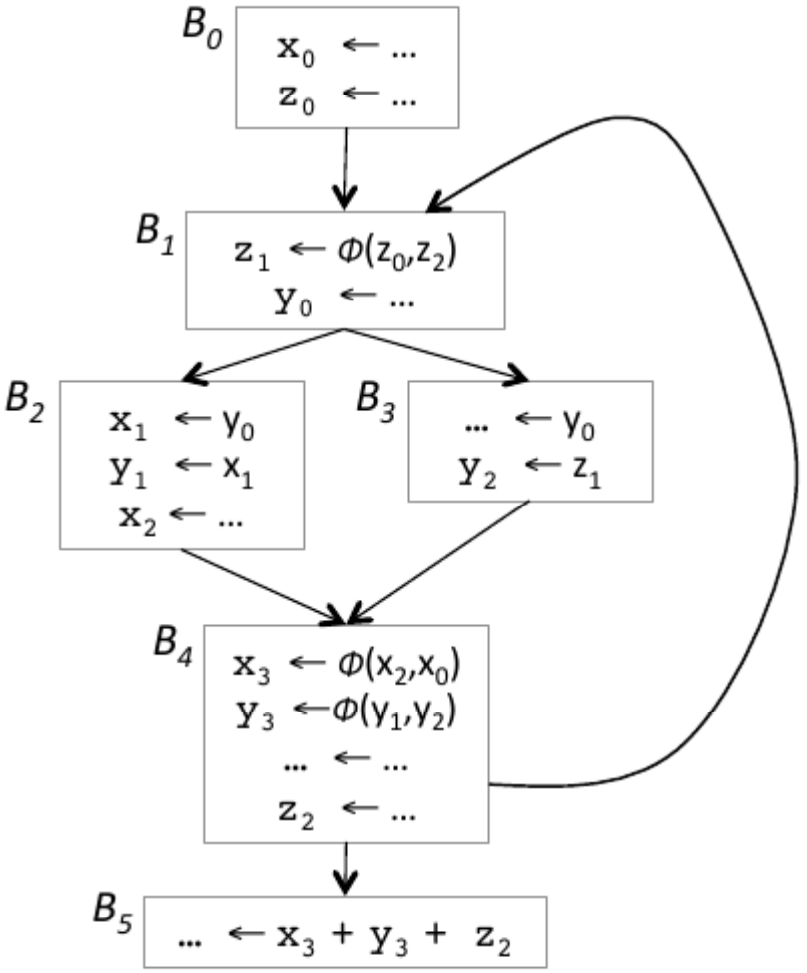
1. {x0},{x1},{x2},{x3}
   {y0},{y1},{y2},{y3}
   {z0},{z1},{z2}
2. … {z0,z1,z2} …
3. … {x0,x2,x3} …
4. … {y1,y2,y3} …

Ergebnis:

{x0,x2,x3},{x1},{y0},{y1,y2,y3}
{z0,z1,z2}

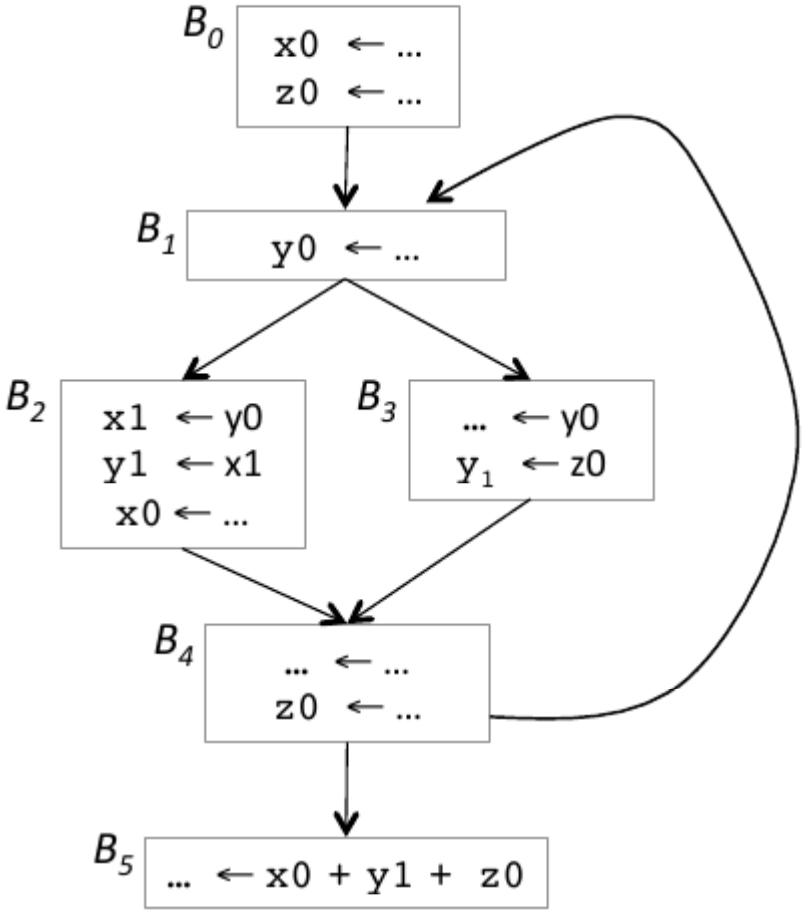
Hinweis: Kopieroperation *x* := y

legt *x* und *y* nicht in selbe Lebenszeit!



Lebenszeiten in SSA-Form 3

- Nun wieder ursprünglichen Nicht-SSA CFG verwenden
- Benenne alle Wertnamen in Lebenszeiten um
  - Vorschlag hier: Ein Element der Menge von SSA-Wertinstanzen
  - {x0,x2,x3}
  - {x1}
  - {y0}
  - {y1,y2,y3}
  - {z0,z1,z2}



LIVE über Lebenszeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Berechne nun LIVE-Mengen
  - Genau wie mit Variablen

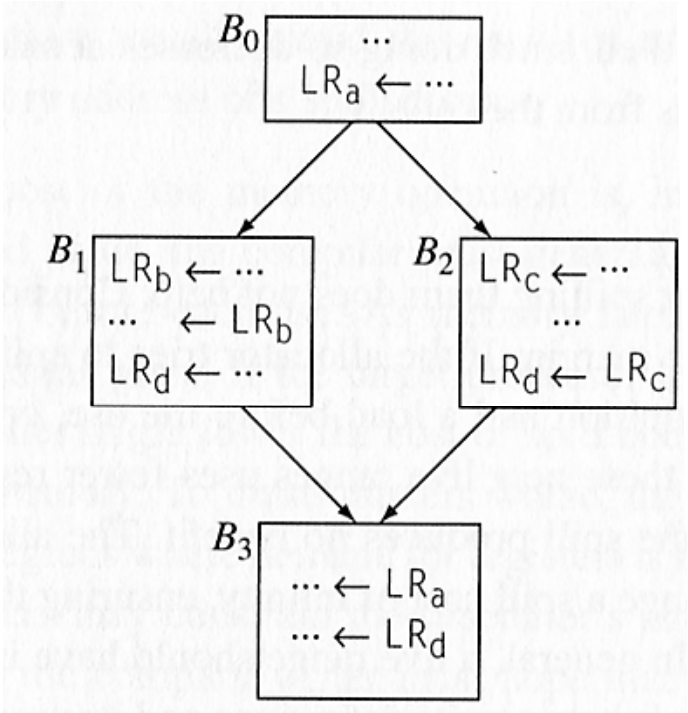
$$\text{LIVEOUT}(b) = \cup_{s \in succ(b)} \text{LIVEIN}(s)$$

$$\text{LIVEIN}(b) = \text{UEVAR}(b) \cup (\text{LIVEOUT}(b) \cap \text{VARKILL}(b))$$

$$\text{LIVEOUT}(n_{final}) = \varnothing$$

- Variablen repräsentieren nun aber **Lebenszeiten von Werten**

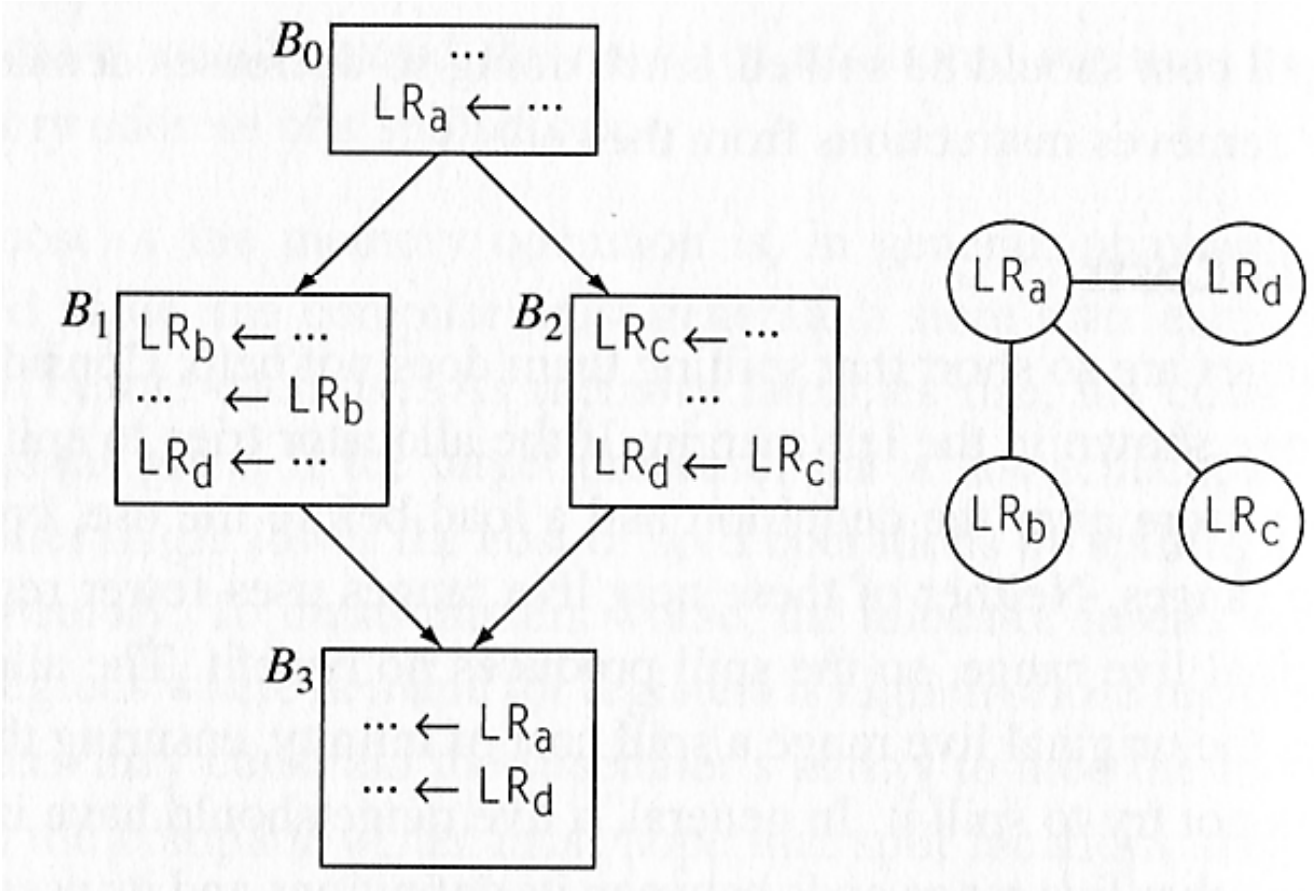
Aufbau des Interferenzgraphen 1




```
for each LR_i
    create a node n_i ∈ N
for each basic block b
    LIVENOW ← LIVEOUT(b)
    for each operation op_n, op_n-1, op_n-2, ...op_1 in b
        with form op_i LR_a, LR_b ⇒ LR_c
        for each LR_i ∈ LIVENOW
            add (LR_c, LR_i) to E
        remove LR_c from LIVENOW
        add LR_a and LR_b to LIVENOW
```

Quelle: EAC2e Fig 13.4 und 13.5

Aufbau des Interferenzgraphen 2



Quelle: EAC2e Fig 13.4

Vorbereitende Überlegungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für *k* Maschinenregister ist eine *k* Einfärbung ausreichend
  - Globales Minimum muss nicht gefunden werden
- Knoten *n* mit weniger als *k* Nachbarn (Grad kleiner k) können immer erfolgreich eingefärbt werden. Notiert als $n° < k$.
  - Wähle eine der Farben, die von den Nachbarn noch nicht benutzt wird

- Genauer: **Spilling**
  - Füge Code hinter jedem DEF ein, um Wert in Speicher zu schreiben
  - Füge Code vor jedem USE ein, um Wert aus Speicher zu lesen
  - Erzeugt sehr viele trivial kleine Lebenszeiten
    - Haben in der Regel weniger Interferenzen mit anderen Lebenszeiten
  - Verwendet üblicherweise *F* reservierte Register

Chaitins Algorithmus zur Registerallokation (PLDI 1982)

TECHNISCHE UNIVERSITÄT DARMSTADT

1 Wähle beliebigen Knoten *n* mit *n*° < *k* und lege ihn auf Stack

2 Entferne *n* und alle seine Kanten aus Interferenzgraph
- Dies kann den Grad benachbarter Knoten auf kleiner als *k* absenken!

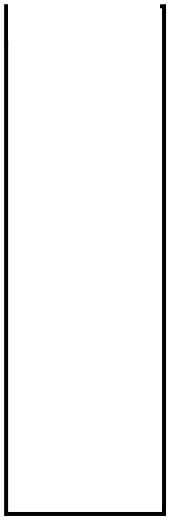
3 Falls nur noch Knoten *n* in Graph mit *n*° >= *k*
- wähle ein *n* (wie?) und merke *n* als zu spillen vor
- Entferne n und seine Kanten aus Interferenzgraph
- Falls nun Knoten *n* mit *n*° < *k* vorhanden, Gehe zu 1; sonst Wiederhole 3

4 Falls nötig: Erzeuge spill code für alle gemerkten Knoten
- Baue Interferenzgraph komplett neu auf und versuche Allokation erneut

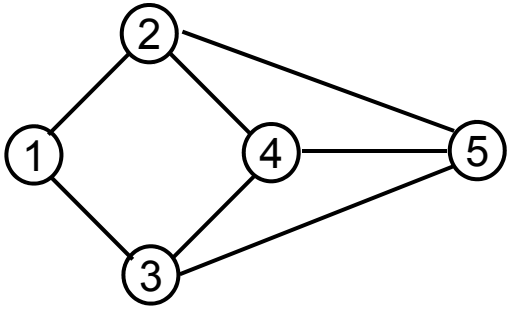
5 Anderenfalls nehme Knoten vom Stack und färbe sie in der Farbe mit dem kleinsten, von den Nachbarn unbenutzten Index

Beispiel 1

3 Register



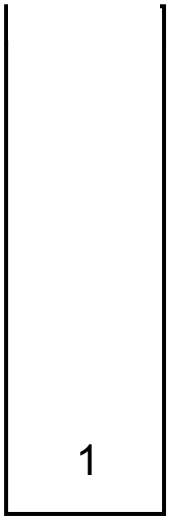
Stack



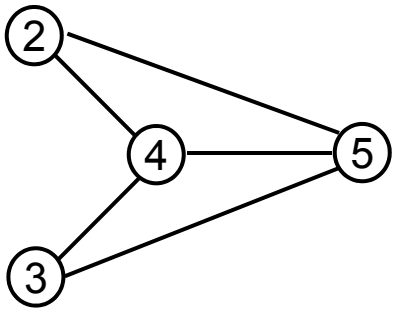
1 ist einziger Knoten mit Grad < 3

Beispiel 2

3 Register



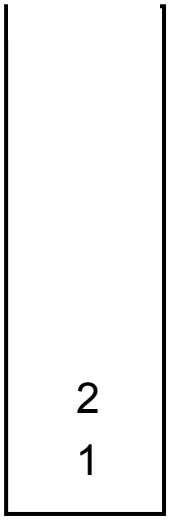
Stack



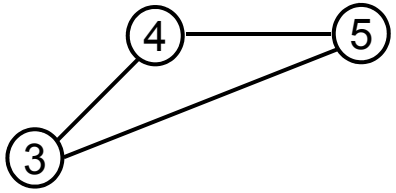
Jetzt haben **2** und **3** einen Grad < 3

Beispiel 3

3 Register



Stack

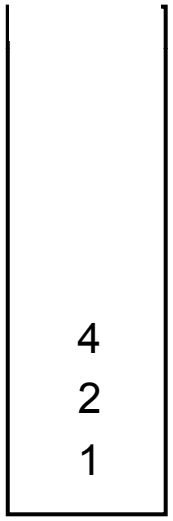


Jetzt haben alle Knoten einen Grad < 3

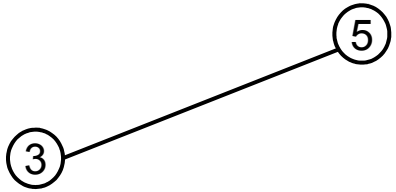
Beispiel 4

TECHNISCHE UNIVERSITÄT DARMSTADT

3 Register



Stack

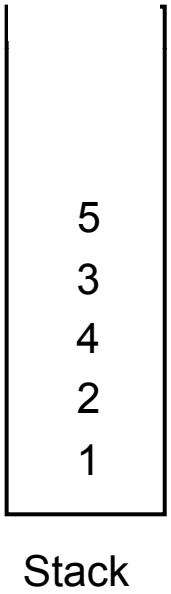


Embedded Systems & Applications

Beispiel 5

TECHNISCHE UNIVERSITÄT DARMSTADT

3 Register



Farben:

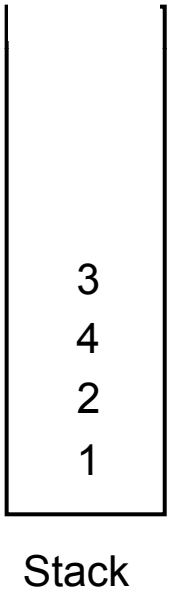
1: 

2: 

3: 

Beispiel 6

3 Registers



Farben:

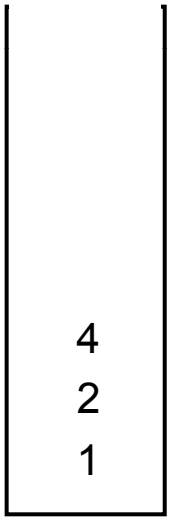
1:

2:

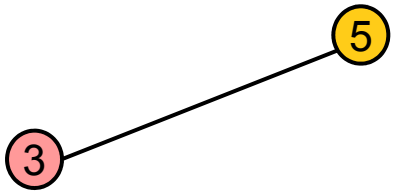
3:

Beispiel 7

3 Register



Stack



Farben:

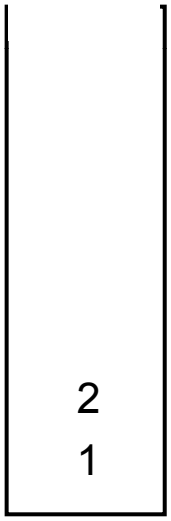
1:

2:

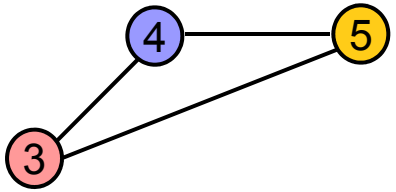
3:

Beispiel 8

3 Register



Stack



Farben:

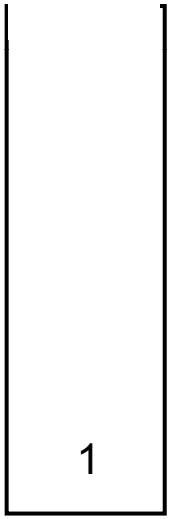
1:

2:

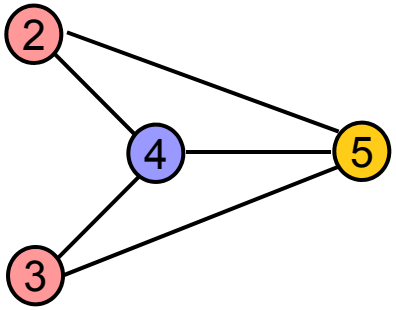
3:

Beispiel 9

3 Register



Stack



Colors:

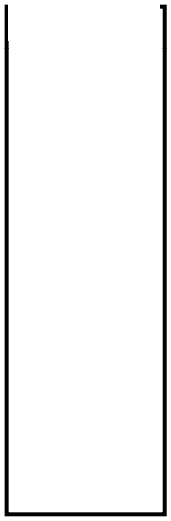
1:

2:

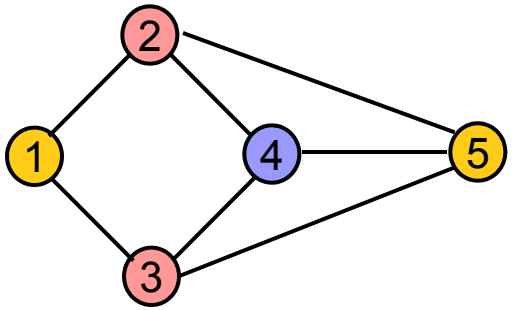
3:

Beispiel 10

3 Register



Stack



Farben:

1:

2:

3:

Diskussion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Chaitins Algorithmus bildete die **Grundlage** für intensive Weiterentwicklungen

- Ein Ansatzpunkt: Chaitins Algorithmus ist **pessimistisch**
  - Wenn nur noch Knoten mit Grad >= *k* vorliegen, sofort zum Spillen vormerken

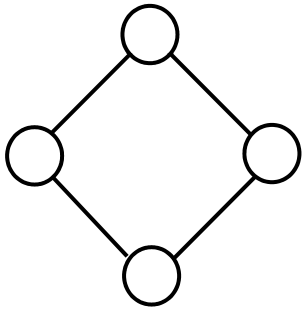
- **Optimistische Einfärbung** (Briggs PLDI 1989)
  - Erstmal weitermachen: Lege Knoten auf Stack wie üblich
  - Möglicherweise ist ja beim Herunternehmen eine Farbe verfügbar

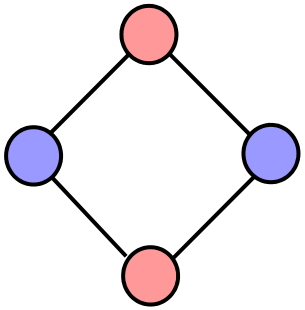
- **Beispiel**
  - Knoten *n* hat *k*+2 Nachbarn, diese benutzen selbst aber nur 1 Farbe!
  - Grad eines Knoten ist nur lose obere Schranke für Einfärbbarkeit

Chaitin vs. Briggs

TECHNISCHE UNIVERSITÄT DARMSTADT

2 Register:





Chaitin entscheidet sofort, einen der Knoten zu spillen

Briggs findet Lösung mit zwei Farben

Chaitin-Briggs Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

TBD SoSe 2014