5. Softwaresynthese

Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

5.1 Übersetzer

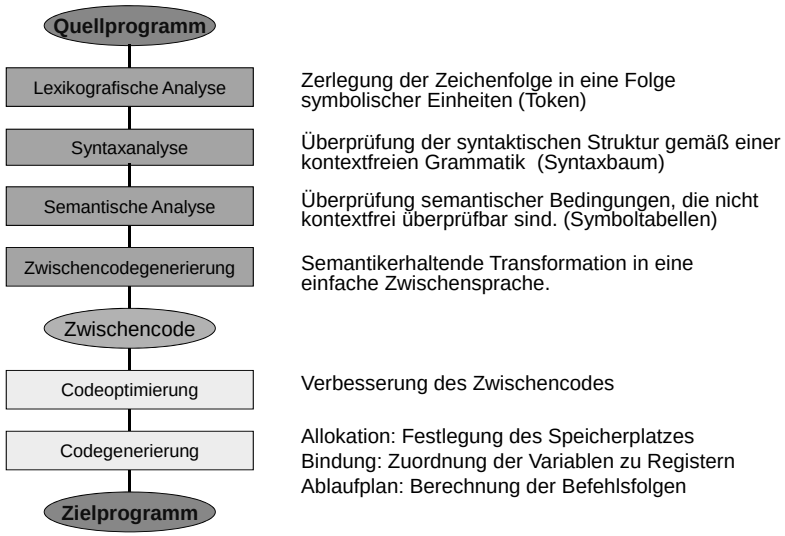
Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

5.1.1 Phasen eines Übersetzers



5.1.2 Zwischencode: 3 Adresscode

Die gebräuchlichste Form einer primitiven Zwischensprache ist der so genannte 3-Adress-Code.

Motivation

- Aufbrechen komplexer Konstrukte der Quellsprache in primitive 3 Adresskonstrukte erfolgt unabhängig von der Zielmaschine durch ein Compiler Front End (→ Portierbarkeit, Retargierbarkeit).
- Zerlegung in Unterprogramme und Einbindung von Libraries. Zwischencode besteht nur noch aus Codeeinheiten für einzelne Routinen mit Aufrufen anderer (einzubindender) Routinen.
- Zwischencode lässt sich leicht in eine Graphdarstellung konvertieren, auf der Bindungs- und Codeerzeugungsprobleme direkt formuliert werden können.

Zwischencode: Beispiel -- Die Sprache „A3“

Folgende einfache Sprache „A3“ kann man als typischen Vetreter der Zwischensprachenebene auffassen:

❶ Eine Übersetzungseinheit ***S*** (Module) besteht stets aus einer Folge von einfachen Statements ***S***: $S_1,\ldots,S_n$

❷ Ein einfaches Statement hat eine der folgenden Formen

- Zuweisung: {<label>:} <ID> = <ID> { <op> <ID> };
- Indizierte Zuweisung: {<label>:} <ID> [< ID>] = <ID>;
  oder {<label>:} <ID> = <ID> [<ID>];
- Verzweigung: {<label>:} if <ID> <relop> <ID> goto <label>;
- Sprung: {<label>:} goto <label>; oder {<label>:} return;
- Parameter: {<label>:} par <ID> ;
- Call: {<label>:} call <module> , <number> ;

❸ Identifier <ID> haben stets die Form gS*n*, tS*n* oder cS*v* wobei *n* eine Zahl und *v* ein Wert ist, S für die Größe (Byte, Short, Word, Double) steht, und g besagt, dass es sich um eine globale Variable handelt. t besagt, dass es sich um eine lokale Variable handelt. c zeigt eine Konstante an.

Beispiel -- Die Sprache „A3“ -- ff

❹ Jedem Statement kann optional eine eindeutige Markierung <label> durch eine Zahl vorangestellt sein.

❺ <op> seien binäre Operatoren, <relop> binäre Vergleichsoperatoren.

Jede Übersetzungseinheit *S* erfülle zudem noch folgende **semantische Bedingungen**:

(L1) Alle vorangestellten Labels (definierende Vorkommen) sind verschieden.

(L2) Jedes in einem Sprung oder einer Verzweigung benutzte Label hat ein definierendes Vorkommen.

(Zw) Alle Operatoren sind kompatibel mit den Größen von Quell- und Zieloperanden einer Zuweisung. Ziele sind niemals Konstanten.

(ID) Globale Variablen müssen über die gesamte Modulbibliothek eindeutig nummeriert sein, temporäre Variablen seien für jede einzelne Übersetzungseinheit durchnummeriert.

Beispiel -- Die Sprache „A3“ -- ff

(UP1) Das <module> eines call-Befehls muss in der Modulbibliothek als Übersetzungseinheit existieren.

(UP2) Einem call-Befehl mit Angabe *n* als <number> müssen genau *n* par-Befehle vorangehen, von denen nur der Erste mit einem Label markiert sein darf. Ist der call-Befehl mit einem Label markiert, muss *n = 0* sein.

(UP3) Operanden von par-Befehlen dürfen nur Variablen sein.

Anmerkungen:

Wir hätten die Sprache noch minimalistischer machen können, indem wir z.B. Konstanten weggelassen hätten. Damit würde man sich aber die Option, immediate Addressierung zu nutzen, sowie eingehendere Datenflussanalyse zu machen, zerstören.

Die explizite Einführung von calls erlaubt es uns, Übersetzungsprobleme auf einzelne Unterprogramme als Grundeinheiten zu beschränken. Jedes Modul schließt implizit mit einem Return ab!

Optimierungsaufgaben der Softwaresynthese

Ausgehend von Zwischencode-Übersetzungseinheiten stellen sich bei der Synthese von Maschinencode folgende Probleme:

- **Registervergabe:** Bindung der „symbolischen Register“ (= Variablen) an die vorhandenen physikalischen Register mit dem Ziel, die Hauptspeicherzugriffe zu minimieren.

- **Codeminimierung**
  Übersetzung des Quellprogramms in einen längenminimalen Maschinencode.

  -- Beide Probleme haben gerade bei eingebetteten Systemen eine besondere Bedeutung, weil sowohl der Datenspeicher als auch der Programmspeicher Kosten verursachen. Datenspeicherzugriffe erniedrigen zudem die Performanz, so dass vor allem in inneren Programmschleifen möglichst Register zu nutzen sind.

5.1.3 Zwischencode als Kontrollflussgraph

Bevor wir die eigentlichen Optimierungsprobleme angehen wollen, sollen Übersetzungseinheiten des Zwischencodes zunächst adäquater dargestellt werden. Dies geschieht in der Regel durch sogenannte Kontrollflussgraphen:

Definition

Gegeben sei eine Folge von Anweisungen $S=(S_1,...,S_n)$. Der Kontrollflussgraph dieser Anweisungsfolge S ist ein gerichteter, kantengeordneter und knotenmarkierter Graph $G_K=(V_K,E_K)$. Zu jeder Anweisung S_i gibt es genau einen Knoten $v_i \in V_K$, der mit S_i markiert ist, und ist wie folgt aufgebaut:

Ist S_i eine Verzweigung, so geht die erste Kante (true) nach S_j, wobei S_j die mit dem Label markierte Anweisung ist, und die zweite Kante nach S_{i+1}.
Ist S_i ein Sprung, so gibt es eine Kante nach S_j, wobei S_j die mit dem Label markierte Anweisung ist.
Sonst gibt es nur eine Kante von S_i nach S_{i+1}.

Beispiel:

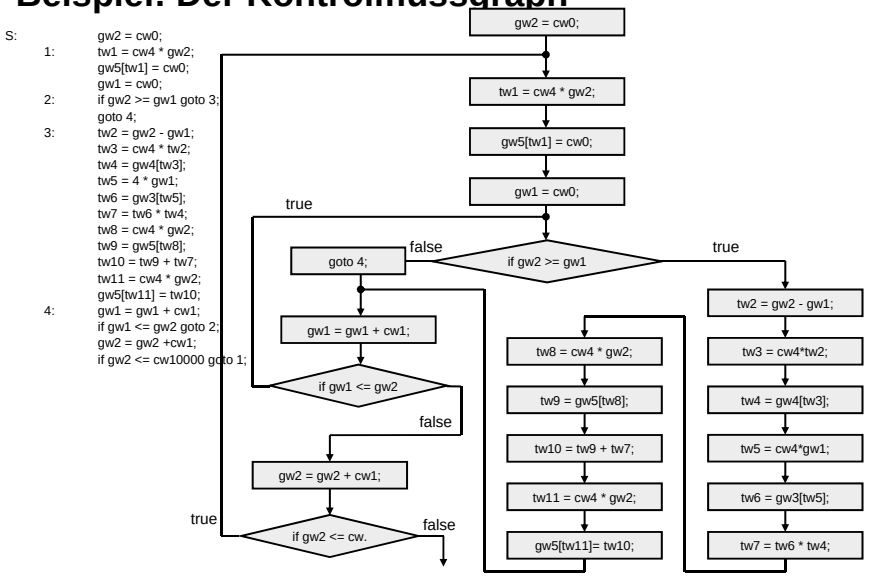
Betrachte folgendes C Programm:

```
int i,n;            // i -> gw1, n -> gw2
int a[];            // a -> gw3
int x[],y[];        // x-> gw4, y -> gw5
main() {
    n = 0;
    do   {
           y[n] = 0;
           i = 0;
           do      {
                     if (n >= i)
                        y[n] = y[n] + a[i] * x[n-i];
                     i = i +1;
                    } while (i <= n)
           n = n+1;
         } while (n <= 10000)
}
```

Das korrespondierende A3-Programm könnte folgendes Aussehen haben:

```
S:      gw2 = cw0;
1:      tw1 = cw4 * gw2;
        gw5[tw1] = cw0;
        gw1 = cw0;
2:      if gw2 >= gw1 goto 3;
        goto 4;
3:      tw2 = gw2 - gw1;
        tw3 = cw4 * tw2;
        tw4 = gw4[tw3];
        tw5 = cw4 * gw1;
        tw6 = gw3[tw5];
        tw7 = tw6 * tw4;
        tw8 = cw4 * gw2;
        tw9 = gw5[tw8];
        tw10 = tw9 + tw7;
        tw11 = cw4 * gw2;
        gw5[tw11] = tw10;
4:      gw1 = gw1 + cw1;
        if gw1 <= gw2 goto 2;
        gw2 = gw2 +cw1;
        if gw2 <= cw10000 goto 1;
```

Beispiel: Der Kontrollflussgraph



5.1.4 Grundblöcke und Grundblockgraph

Definition

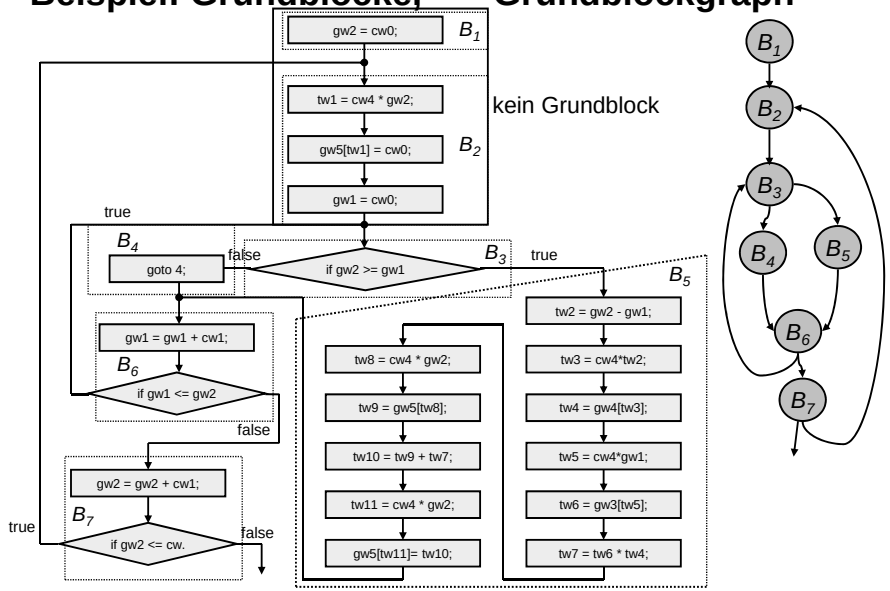
Ein **Grundblock** bezeichnet eine maximal lange Folge fortlaufender Anweisungen $S_i,\ldots,S_{i+b}$, mit:

- für jedes $0 < j \leq b$: ist (S_k, S_{i+j}) Kante des Kontrollgraphen, dann gilt $k = i+j-1$.
- für jedes $0 \leq j < b$: S_{i+j} ist kein call-, Sprung- oder Verzweigungsbefehl.

Anmerkung:

Die meisten Optimierungs- und Generierungsverfahren arbeiten auf Grundblöcken. Daher ist eine Zerlegung in Grundblöcke naheliegend. Durch Verschmelzen von Grundblöcken zu Knoten erhält man den sog. **Grundblockgraphen** aus dem Kontrollflussgraphen. Diese Zerlegung lässt sich ausgehend von der Quelle S_1 mit elementaren Graphalgorithmen leicht berechnen.

Beispiel: Grundblöcke, Grundblockgraph



Übersetzung von Grundblöcken in Taskgraphen

Im Grunde kann man nun Grundblock für Grundblock in einen DAG übersetzen und könnte so Tasksystem für Tasksystem Hardware generieren, die über ein Steuerwerk, das aus dem Grundblockgraphen abgeleitet wird, koordiniert wird:

Gegeben sei ein Grundblock von Anweisungen $S_1,...,S_n$. Der Taskgraph dieser Anweisungsfolge S ist ein gerichteter, azyklischer Graph $G_S=(V_S,E_S)$, der wie folgt aufgebaut wird: $V_S = \{\}$, $E_S = \{\}$.

Bearbeite die Statements der Reihe nach wie folgt:

Ist S_i eine Zuweisung der Form „z = x op y“ oder „z = x[y]“, dann

- prüfe, ob es eine Task *v* mit dem Operationstyp „op“, bzw. „[ ]“ gibt, deren beide Vorgänger mit *x,y* markiert sind.
  - Falls ja, markiere *v* mit „z“,
  - Falls nein, erzeuge einen Knoten *v* mit Operationstyp „op“ bzw. „[]“, und markiere ihn mit „z“.
    - Falls ein mit „x“ markierter Knoten *u* existiert, führe eine Kante *(u,v)* als erste Kante ein,
    - sonst kreiere eine Quelle mit Operationssymbol und Marke „x“.
    - Verfahre für die zweite Kante analog.

Übersetzung von Grundblöcken in Taskgraphen

- Falls schon ein mit „z“ markierter Knoten *u* existiert, führe eine Kante *(u,v)* als dritte Kante ein, sonst keine dritte Kante. Lösche die Marke „z“ am Knoten *u*.

Ist S_i eine Zuweisung der Form „z[x] = y“ dann

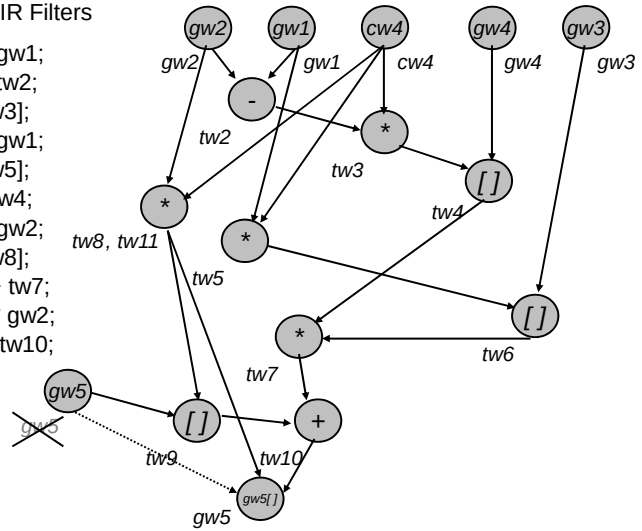
- erzeuge einen Knoten *v* mit Operationstyp „z[ ]“, und markiere ihn mit „z“.
  - Falls ein mit „x“ markierter Knoten *u* existiert, führe eine Kante *(u,v)* als erste Kante ein,
  - sonst kreiere eine Quelle mit Marke „x“.
  - Verfahre für die zweite Kante analog.
- Falls schon ein mit „z“ markierter Knoten *u* existiert, führe eine Kante *(u,v)* als dritte Kante ein, sonst keine dritte Kante. Lösche die Marke „z“ am Knoten *u*.

Die Löschmaßnahme sorgt dafür, dass Datenabhängigkeiten immer vom letzten Schreibzugriff ausgehen. Sobald *z* wieder als Operand benutzt wird, wird *z* automatisch datenabhängig vom letzten Schreibzugriff. Ferner muss *z* datenabhängig von seinem vorletzten Schreibzugriff sein.

Beispiel: Übersetzung von Grundblöcken

Wir betrachten folgenden Grundblock des FIR Filters

```
tw2 = gw2 - gw1;
tw3 = cw4 * tw2;
tw4 = gw4[tw3];
tw5 = cw4 * gw1;
tw6 = gw3[tw5];
tw7 = tw6 * tw4;
tw8 = cw4 * gw2;
tw9 = gw5[tw8];
tw10 = tw9 + tw7;
tw11 = cw4 * gw2;
gw5[tw11] = tw10;
```



5.2 Registervergabe und Registerbindung

Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

Vorbemerkungen

Im Rahmen der Softwaresynthese stellt sich das Problem der Registerbindung etwas allgemeiner als bei der Hardwaresynthese:

- Die Registerzahl ist fest vorgegeben und in der Regel zu klein, um alle lebendigen Variablen aufzunehmen.
  - ➔ Finde eine Vergabe mit minimaler Zahl von Hauptspeicherzugriffen.
- Registervergabe kann lokal, d.h. auf Grundblöcken, oder „global“, d.h . für innere Schleifen, betrachtet werden.
- Die Konfliktgraphen sind nicht notwendig Intervallgraphen!
- Wir können dennoch, zumindest teilweise, von unseren Graphfärbungstechniken aus Kapitel 4 profitieren.

5.2.1 Lebensspannen

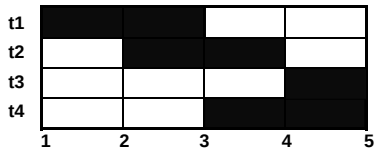
Wir benötigen wieder die Lebensspanne der symbolischen Register im 3-Adress-Code.

Beispiel

Zwischencode

```
1: t1=5;
2: t2=3*20;
3: t4=t1*t2;
4: t3=2*t2;
  // nur t3 und t4 seien an
  // dieser Stelle noch aktiv
```

Aktivitäts- oder **Lebensspannen**



Werte müssen solange im Speicher / Register gehalten werden, wie sie noch nachfolgend verwendet werden.

Berechnung der Lebensspannen

Definition

Sei $B = S_i,\ldots,S_k$ ein Grundblock. Die Lebensspannen einer Variablen *x,* ist gegeben durch eine Menge *L(x)* von paarweise disjunkten Intervallen *[l,r)* mit $i \le l < r \le k + 1$.

Eine Variable *x* heißt aktiv zum Blockeintritt, wenn sie vor ihrer ersten Zuweisung im Block als Operand gelesen wird. Erfolgt an *x* keine Zuweisung im ganzen Block, so ist *x* aktiv zum Blockeintritt, wenn *x* zum Blockende aktiv ist.

Eine Variable *x* heißt aktiv zum Blockende, wenn es im Grundblock-graphen einen von *B* aus erreichbaren Nachfolgeblock *B‘* gibt, in dem x aktiv zum Blockeintritt ist.

Die Eigenschaft, aktiv zum Eintritt und Ende zu sein, kann durch DFS auf dem Grundblockgraphen und Durchmustern der Blöcke offenbar einfach berechnet werden.

Berechnung der Lebensspannen

Sei $B = S_i,\ldots,S_k$ ein Grundblock. Die Lebensspannen einer Variablen x berechnen sich nun wie folgt:

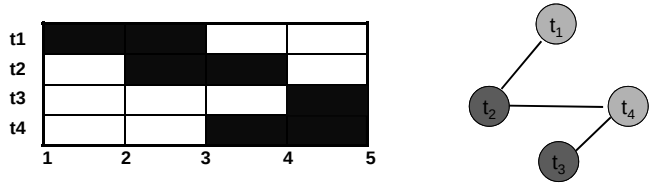
if x aktiv zum Blockeintritt,
  then setze $l = i$; und markiere *x als aktiv*;
  else setze $l = 0$; markiere *x als nicht aktiv*;
for j = i to k
do if x ist Operand in S_j then setze $r = j$; markiere *x als aktiv*;
  if S_j hat die Form „x =…“
    then if x aktiv then $L(x) = L(x) \cup \{ [l,r) \}$; $l = j$;
      else $l = j$;
    markiere *x als nicht aktiv*;
od;
if x aktiv zum Blockende then $L(x) = L(x) \cup \{ [l,k+1) \}$;
  else if x aktiv then $L(x) = L(x) \cup \{ [l,r) \}$;

Man erkennt, dass die Menge $L(x)$ nicht notwendig einelementig ist.

5.2.2 Lokale Registervergabe durch Graphfärbung

Definition

Ein Registerkonfliktgraph $G=(V,E)$ eines Grundblocks B ist ein ungerichteter Graph, in dem die Knotenmenge V die Menge der Variablen in B darstellt. Die Kantenmenge E drückt die Konfliktrelation aus, d.h. $(v_i,v_j) \in E$ bedeutet, dass es Intervalle $[l_i,r_i) \in L(v_i)$ und $[l_j,r_j) \in L(v_j)$ gibt, mit $[l_i,r_i) \cap [l_j,r_j) \neq \{\}$.

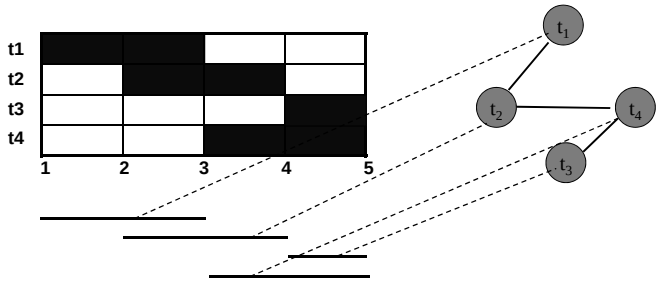


Vergabe durch Graphfärbung

Algorithmen zur Registervergabe

Beobachtung

Ist der Registerkonfliktgraph $G=(V,E)$ ein Intervallgraph, dann kann er mit dem Leftedge-Algorithmus optimal gefärbt werden, falls genügend Register zur Verfügung stehen.



Registervergabe unter Ressourcenbeschränkung

In der Regel haben wir aber eine Ressourcenbeschränkung von k Registern, und keinen Intervallgraphen. Hier kann man auf folgende Heuristik zurückgreifen:

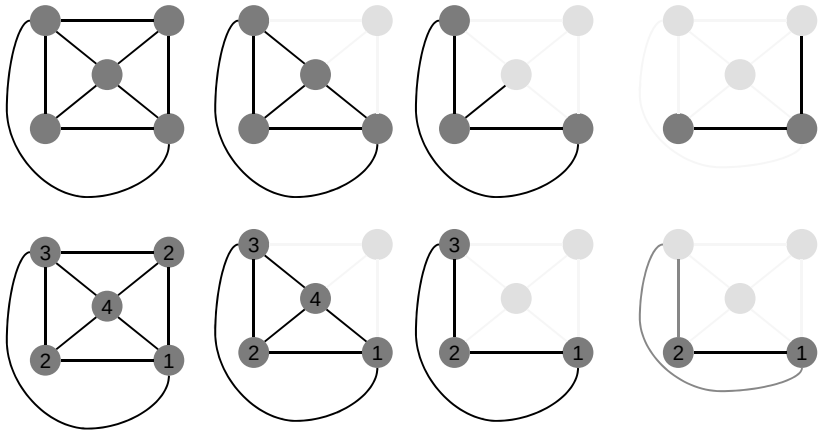
Verfahren von Chaitin ['82]

- Entferne der Reihe nach alle Knoten $v \in V$ mit $degree(v) < k$ und die dazugehörigen Kanten. Diese Knoten sind mit k Farben färbbar unabhängig von der Färbung der übrigen Knoten.
  -- Man färbe den Restgraphen und füge den Knoten v wieder ein, indem man ihm eine Farbe gibt, die keiner seiner $< k$ Nachbarn besitzt.
- Wird der Graph leer, so ist der Graph mit k Farben färbbar.
- Wird der Graph nicht leer, so haben alle Knoten im Restgraphen Grad $\geq k$. Wähle heuristisch einen Knoten v aus und "entferne geeignete Kanten von v aus dem Graphen"

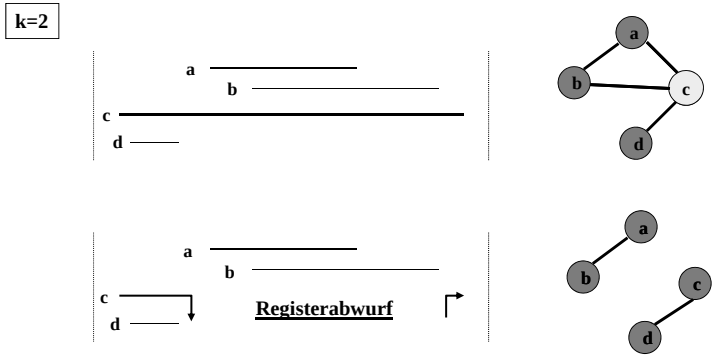
→ **Registerabwurf**, d.h. das entsprechende „symbolische Register“ wird zwischenzeitlich im Hauptspeicher und nicht in einem Register gespeichert. Erfordert Einfügen von Store/Load Befehlen.

Beispiel:

Ist folgender Graph 4-färbbar? 1 2 3 4



"Entfernung von Kanten"



5.2.3 Globale Registervergabe

Erfahrungsregel

Programme verbringen die größte Ausführungszeit in inneren Schleifen $\rightarrow$ Konzentration der Registervergabe auf Schleifen

Beispiel

```
for (i:=1; i≤n; i++)
{
1:  x1:=x3*10;
2:  x2:=x4+20;
3:  x3:=x1+5;
4:  x4:=x2+x3;
}
```

Grundblock

Alleine den Grundblock für sich zu betrachten, reicht nicht aus.

Mehrere Iterationen müssen gleichzeitig betrachtet werden, um eine effiziente Registervergabe berechnen zu können

Schleifen

Definition

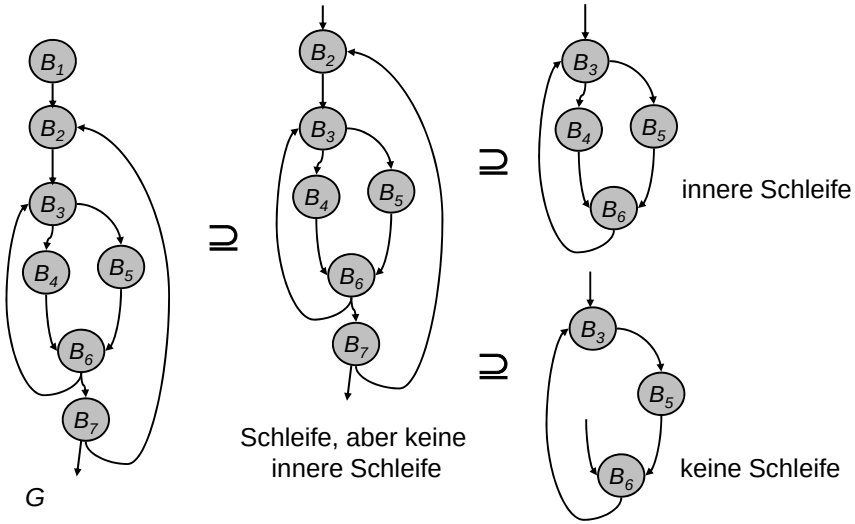
Sei $G= (V,E)$ ein Grundblockgraph einer Übersetzungseinheit S. Ein Teilgraph $G'= (V',E')$ von G heißt **Schleife**, wenn G' stark zusammenhängend ist, und es einen eindeutigen Eingangsknoten v^* gibt, so dass jeder Pfad von einem Knoten $u \in V \backslash V'$ nach $v \in V'$ über v^* geht.

G' heißt **innere Schleife**, wenn G' selbst keine Schleife von G als echten Teilgraphen enthält.

Schleifen und innerste Schleifen können durch iterative Anwendung von DFS (starke Zusammenhangskomponenten, vgl. Rechnergestütztes Layout WS07/08) bestimmt werden.

Schleifen: Beispiele

Betrachte folgenden Grundblockgraphen



Registervergabe in Schleifen

Die einfachste Frage, die man im Zusammenhang mit Schleifen angehen kann, ist die der Registerzuteilung:

Wir nehmen an, wir hätten einen Zielmaschinenbefehlssatz mit Speicheroperanden, einen kleinen Vorrat von Registern zum Speichern von Zwischenergebnissen, und einen gewissen Vorrat an Registern, die wir fest an Variablen binden können.

Ziel:

Bestimme eine Teilmenge der in der Schleife auftretenden Variablen, die an Register gebunden werden, so dass die Zahl der Speicherzugriffe möglichst gering wird.

Dazu kann man ein einfaches heuristisches Maß für den Gewinn bei Bindung an ein Register angeben:

Registervergabe in Schleifen

Für x und einen Grundblock B_i sei

$V(x,B_i)$ = Zahl der Vorkommen von x als Operand im Block B_i vor der ersten Zuweisung an x in B_i

$A(x,B_i) = 1$, falls x aktiv beim Austritt aus B_i ist und eine Zuweisung in B_i hatte, sonst 0.

Dann spart man, wenn x in der Schleife an ein Register gebunden wird, $V(x,B_i)$ Speicherzugriffe. Nach der ersten Zuweisung an x gehen wir optimistisch davon aus, dass bei den übrigen Auftreten der Wert in einem temporären Register ebenfalls vorliegt und zählen diese deshalb nicht mit.

Ist x aktiv bei Verlassen von B_i , dann erspart man sich das Wegspeichern von x, und ein späteres wieder Laden, da x im Register vorgehalten wird.

Wir setzen daher den Gewinn von x für eine Schleife mit Blockmenge V' fest als

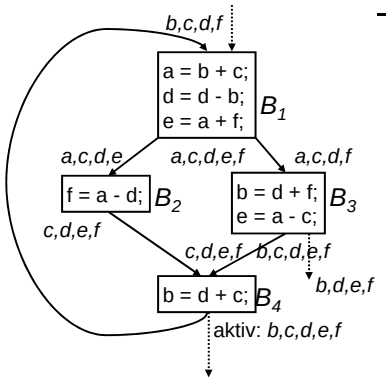
$$\Delta_x := \sum_{B_i \in V'} V(x,B_i) + 2 \cdot A(x,B_i)$$

Registervergabe in Schleifen

Einfache Heuristik von Aho et. al.:

Teile den Variablen x mit dem größten Gewinn Δ_x die Register zu.

Beispiel: Folgende Schleife



V	a	b	c	d	e	f
B_1	0	2	1	1	0	1
B_2	1	0	0	1	0	0
B_3	1	0	1	1	0	1
B_4	0	0	1	1	0	0

A	a	b	c	d	e	f
B_1	1	0	0	1	1	0
B_2	0	0	0	0	0	1
B_3	0	1	0	0	1	0
B_4	0	1	0	0	0	0

Δ	a	b	c	d	e	f
	4	6	3	6	4	4

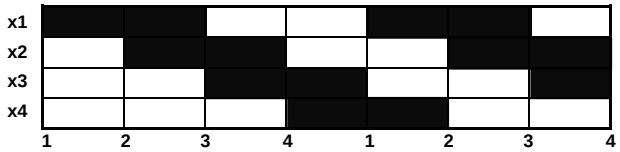
3 freie Register:
Halte etwa *b,d,a* in Registern

Iterationsübergreifende Lebensspannen

Beispiel

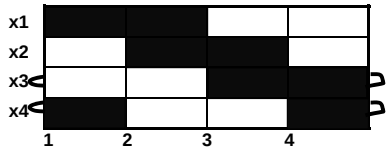
```
for (i:=1; i≤n; i++)
{
1:  x1:=x3*10;
2:  x2:=x4+20;
3:  x3:=x1+5;
4:  x4:=x2+x3;
}
```

Beobachtung:
Beschränkt man sich auf Schleifen, die nur einen einzigen Grundblock als Rumpf haben, so erhält man zirkulare Lebensspannen. Im einfachsten Falle müssen wir also wieder einen Graphen mit zirkularen Kanten färben (vgl. 3.1).



Lebenspannen über mehrere Iterationen

Graph mit zirkularen Kanten



Definition

Die Dichte *d(G,t)* eines Registerkonfliktgraphen *G* mit zirkularen Kanten zu einem Zeitpunkt *t* ist die Anzahl sich schneidender Abschnitte zum Zeitpunkt *t*.

Weiter sei $d_{max}(G)$:=max{ $d(G,t)$; $t \in$ N } und

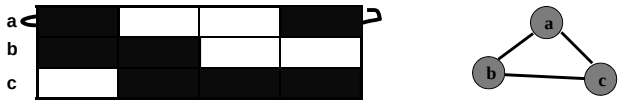
$d_{min}(G)$:=min{ $d(G,t)$; $t \in$ N }

Beobachtung

Nicht jeder Registerkonfliktgraph mit zirkularen Kanten ist mit $d_{max}(G)$ färbbar!

Färbung bei zirkularen Kanten

Beispiel: Es gilt $d_{max}(G)=2$, aber man braucht wenigstens 3 Register!



Satz -- Tucker

Sei *G* ein Registerkonfliktgraph mit zirkularen Kanten. *G* ist optimal färbbar mit *k* Farben für ein $k \in$ N mit

$$d_{max}(G) \le k \le d_{max}(G)+d_{min}(G)$$

Beweis: Man erhält die Schranke durch Anwendung des Sort&Match Algorithmus aus Kapitel 3.1.

Heuristik von Hendren (fat spots) [1992]

Definition

Die Menge der **fetten Punkte** eines Registerkonfliktgraphen *G* mit zirkularen Kanten ist die Menge aller Zeitpunkte *t* mit $d(G,t)=d_{max}(G)$.

Definition

Eine **Überdeckung der fetten Punkte** (fette Überdeckung) eines Registerkonfliktgraphen *G* relativ zu einem Intervall v_i, ist eine Teilmenge *Y* von Intervallen mit

- $v_i \in Y$
- $\forall v_k, v_m \in Y$: $v_k \cap v_m = \{\}$
- Zu jedem fetten Punkt *t* gibt es ein Intervall *v* in *Y*, das *t* enthält.

Beobachtung

1. Die Intervalle einer Überdeckung der fetten Punkten relativ zu einem Intervall *v* stehen nicht zueinander in Konflikt, können also mit der gleichen Farbe gefärbt werden.
2. Entfernt man die Knoten einer Überdeckung der fetten Punkte relativ zu einem Intervall aus dem Registerkonfliktgraphen, so erniedrigt sich die maximale Dichte um 1.

Die Heuristik

Phase 1

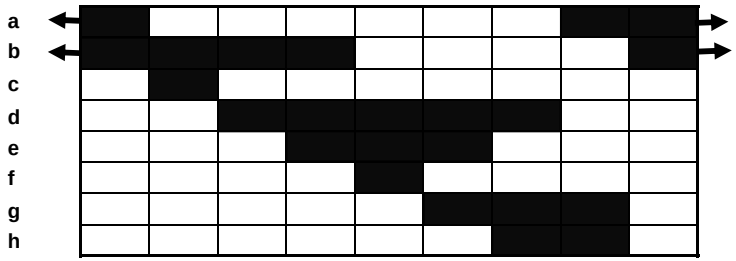
- ⊙ Bestimme die fetten Punkte des Registerkonfliktgraphen
- ⊙ Bestimme eine Überdeckung *Y* relativ zu einem zyklischen Intervall
- ⊙ Weise den Intervallen aus *Y* eine noch nicht benutzte Farbe zu
- ⊙ Lösche die Intervalle von *Y* aus dem Registerkonfliktgraphen
- ⊙ Wiederhole Phase 1 solange es noch zyklische Intervalle gibt

Phase 2

- -- Nun ist der Graph nur noch ein Intervallgraph !
- ⊙ Färbe die restlichen Intervalle mit dem Leftedge-Algorithmus

Illustration der Heuristik von Hendren

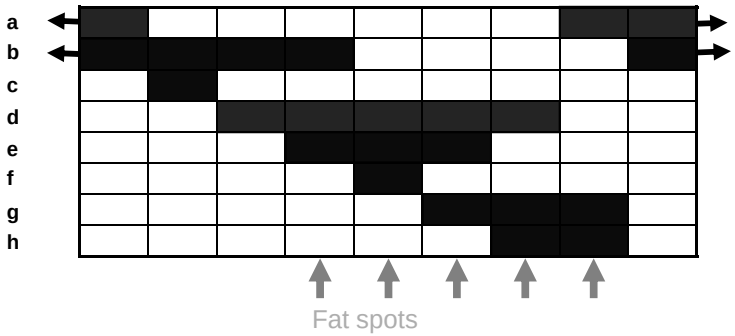
Die Problemstellung *G*:



2 zirkulare Kanten, $d_{max}(G)=3$, $d_{min}(G)=2$

Illustration der Heuristik von Hendren

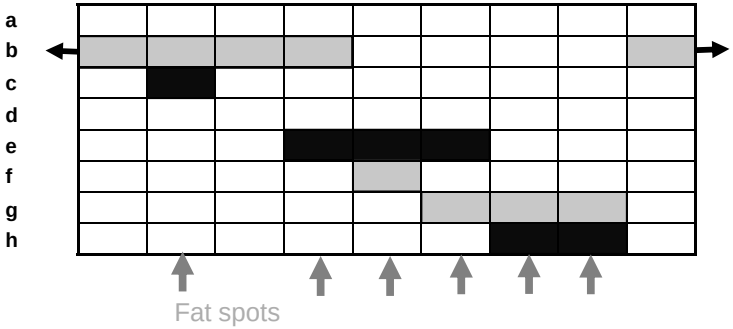
Fette Überdeckung relativ zu Knoten *a*:



- Alle **fetten Punkte** müssen von der Überdeckung überdeckt sein
- Das Intervall *a* muss in der Überdeckung sein
- Die Intervalle der Überdeckung müssen paarweise disjunkt sein

Illustration der Heuristik von Hendren

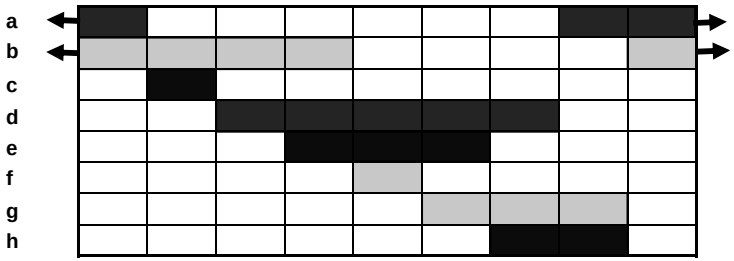
Fette Überdeckung relativ zu Knoten *b*:



- Die fetten Punkte müssen neu berechnet werden: $d_{max}(G_{neu})=2$

Illustration der Heuristik von Hendren

Restintervalle (keine zirkularen Kanten mehr vorhanden)



- Es gilt nun $d_{max}(G_{neu})=1$, d.h. die restlichen Intervalle können mit der gleichen Farbe gefärbt werden.

Anmerkungen zur Heuristik von Hendren

Lemma
Ist Phase 1 erfolgreich, so ist nach Phase 1 die Dichte des restlichen Graphen nur noch $d_{max}(G) - m$, wobei m die Anzahl der zirkularen Kanten im ursprünglichen Registerkonfliktgraphen G bezeichnet.

Beweis: Da wir **alle** fetten Punkte überdecken, sinkt die maximale Dichte in jedem Schritt um 1.

Korollar
Ist Phase 1 erfolgreich, so färbt die Heuristik von Hendren den Registerkonfliktgraphen mit minimal vielen Farben, d.h. mit $d_{max}(G)$ vielen Farben.

Beweis: Da die Abschnitte disjunkt gewählt wurden, wurden in Phase 1 nur m Farben verbraucht. Leftedge färbt nun den Rest in $d_{max}(G) - m$ Farben.