Kapitel 11

Registerzuteilung

Kapitel 11: Registerzuteilung

Registerzuteilung (engl. Register Allocation)

$$
\begin{aligned}
x &\leftarrow \mathrm{param0} \\
y &\leftarrow \mathrm{param1} \\
t_0 &\leftarrow \mathrm{add}\ x, y \\
t_1 &\leftarrow \mathrm{mul}\ t_0, y \\
&\vdots
\end{aligned}
$$

Gegeben: Programm mit angeordneten Befehlen der Zielmachine. (Temporäre) Variablen als Operanden.
Problem: Annahme während der Optimierungsphase: Anzahl verfügbarer Register ist unbeschränkt – Es werden beliebig viele Variablen benutzt.
Aufgabe: Reduktion auf die tatsächlich verfügbaren, endlich vielen Register.
Prinzip: Teile Variablen Register auf den verschiedenen Registerklassen zu. Wenn nötig lagere Werte in den Hauptspeicher aus.

Registerklassen

Register können nicht beliebig verwendet werden:

- Gebrauch der Register durch Hardware festgelegt:
  - Gleitpunkt-/Integer Register
  - Spezielle Register: Befehlszähler, Bedingungsanzeige
  - Adressregister
  - ...
- Gebrauch durch Konventionen des Laufzeitsystem festgelegt
- Dringend benötigte Werte (z.B. Kellerpegel) können nicht in den Speicher ausgelagert werden.

Registerklassen:

Teile Register in Klassen mit ähnlichen Beschränkungen ein. Typisch: Integer, Fließkomma, Spezielle Register (Rahmenzeiger, Bedingungsanzeige). Zuteilungsverfahren betrachten die Klassen separat.

Registerzuteilung: Wann/Wo

„Wann”:

- Nach Codeselektion; Probleme:
  - Kosten in Codeselektion von Registerzuteilung abhängig,
  - Auslagerungscode (*spill-code*) von Registerzuteilung abhängig,
  - Bestimmter Code nur mit bestimmten Registern auswählbar.
- Vor Codeselektion; Probleme:
  - Codeselektion definiert Anzahl der benötigten Register
  - Manche Werte werden nie explizit berechnet, z.B. Werte auf Adressierungspfaden
- Codeselektion – Registerzuteilung – Codeselektion
- Während der Codeselektion (*on the fly*)
- Aber Lebendigkeit nur definiert nach Anordnung der Befehle

„Wo”:

- Ausdrücke
- Grundblöcke (lokal)
- Schleifen
- Prozeduren (global)
- Programme

Registerzuteilung – Aufgaben

- Aufgabe der Registerzuteilung ist Abbildung der Programmvariablen auf Prozessorregister
- Aufgaben im Detail:

  Zuteilen Finde Abbildung von Programmvariablen auf Prozessorregister

  Auslagern Lagere Variablen in den Hauptspeicher aus, falls nicht genug Register verfügbar sind

  Verschmelzen Eliminiere unnötiges Kopieren von Variablen im Programm (Kopien mit gleichm Quell- und Zielregister entfernen)

  Beschränkungen Stelle sicher dass Beschränkungen für die Verwendung der Register eingehalten werden

Verfahren zur Zuteilung von Registern

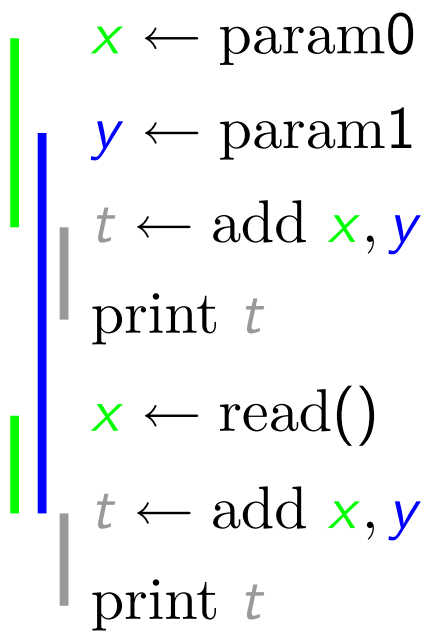
- Lokale Registerzuteilung mit Freiliste (*on the fly*)
- *linear-scan register allocation*
- *Graphfärben*

Lebendigkeit

Definition:

Eine Variable heißt lebendig wenn der in in ihr enthaltene Wert (möglicherweise) später gelesen wird.

Beispiel: Lebendigkeit für x, y, t



Berechnung, Interferenz

Berechnung

- Lokal: Durchlaufe Grundblock rückwärts (von Ende zum Anfang):
  - Bei Verwendung wird Variable lebendig.
  - Bei Definition „stirbt” Variable.
- Global: Datenflußanalyse nötig (hier nicht behandelt).

Interferenz 2 Variablen interferieren wenn sie an einem Programmpunkt beide lebendig sind. Interferieren 2 Variablen so müssen sie unterschiedliche Registern zugeteilt bekommen!

Kapitel 11: Registerzuteilung

Lokale Registerzuteilung mit Freiliste (*on the fly*)

- Bestimme letzte Verwendungen von Werten im Grundblock.
- Durchlaufe Grundblock von Anfang bis Ende:
  - Falls Register benötigt wird rufe *allocReg*() auf; nach letzter Verwendung *freeReg*(r).
  - *allocReg*(): Gibt ein freies Register zurück, falls keines mehr frei ist, löse Ausname aus. Entferne Register aus der Freiliste.
  - *freeReg*(r): Füge Register r in die Freiliste ein.
- Am Ende des Grundblocks (teilweise auch Ausdrucks) werden alle Variablen in den Speicher geschrieben.
- Achtung: Falls Register fehlen, kann kein Programm erzeugt werden (die ersten Turbo Pascal Übersetzer funktionierten wirklich so), ggf. muss dieses Verfahren um die Möglichkeit des Auslagerns erweitert werden.

Erweiterungen

Oft Kombination von lokalen mit globalen Methoden:

- Teile Variablen deren Lebenszeiten komplett innerhalb eines Grundblocks liegen mit lokalem Verfahren zu.
- Teile übrige Variablen mit globalem Verfahren zu. Beachte dabei Interferenzen mit bereits lokal vergebenen Registern.
- Lokales Verfahren kann oft mit Codeauswahl kombiniert werden.

Nutze höhere Geschwindigkeit/besseres Auslagerungsverhalten für lokale Variablen ohne, dass Werte an Grundblockgrenzen zurück in den Speicher geschrieben werden.

Kapitel 11: Registerzuteilung

linear-scan register allocation

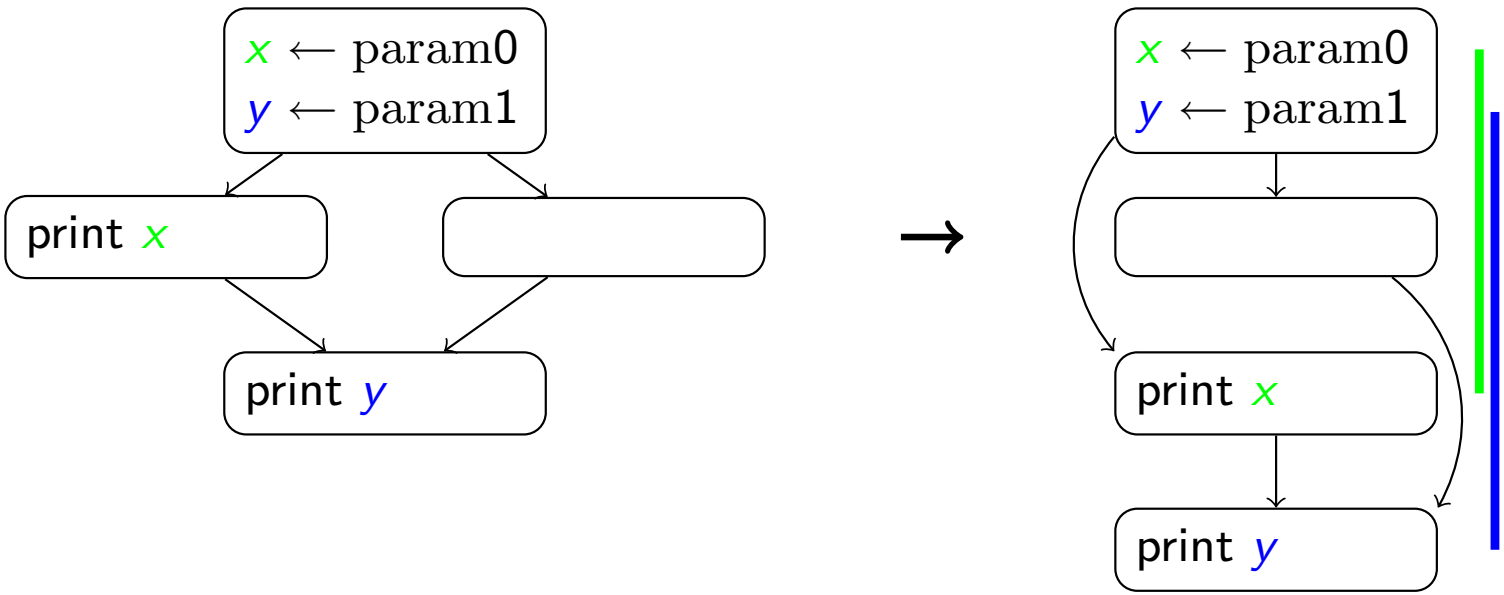
- Die Laufzeit der Graphfärbung ist (sehr) hoch.
- Codequalität von rein lokalen Verfahren schlecht.
- Linear-scan register allocation ist eine Erweiterung des *on the fly* Ansatzes.

Algorithmus

- Berechne Lebendigkeitsinformation und Ablauflaufgraph.
- Durchlaufe das Programm in umgekehrter Postfixordnung: Dies erzeugt linear Liste aller Grundblöcke.
- Berechne Lebendigkeitsinterval für jede Variable.
- Durchlaufe sortierte Intervalliste:
  - Verwende *allocReg*() und *freeReg*() wie beim *on the fly* Ansatz.
  - Auslagern bei Bedarf. Lagere längste verbleibende Intervalle zuerst aus.

linear-scan – Linearisierung

Eine Linearisierung kann Grundblöcke unnötigerweise überdecken.



linear-scan – Erweiterungen

Es existieren verschiedene Verbesserungen um Schwächen des original Linear-Scan Ansatz zu beheben:

- Mehrere Intervalle pro Variablen: Ausnutzen von „Lücken” in den Lebenszeiten.
- Handhabung von Register-Constraints
- Kein Freihalten von Registern für Reloads; erzeuge stattdessen neue Intervalle
- Splitten von Intervallen

Kapitel 11: Registerzuteilung

Registerzuteilung mit Graphfärbung (nach Chaitin)

Prinzip (Chaitin 1981):

- Konstruiere für jede Prozedur einen Interferenzgraph
- Knoten sind die Variablen (auch temporäre) des Programms
- Knoten e, e' durch Kante verbinden, wenn sie nicht gleichzeitig dasselbe Register belegen können.
  - Grund von Unverträglichkeit: überlappende Lebensdauer.
  - Information: Definition und Benutzung von Werten
- Graphfärbung mit minimaler Farbanzahl (*chromatische Zahl* $\chi(G)$) liefert die Minimalanzahl benötigter Register und gleichzeitig die Registerzuordnung.

Algorithmus

- Ist Register-Interferenz-Graph mit k (=Anzahl der Register) färbbar?
  Aber: Bestimmung der chromatischen Zahl ist *NP-Problem*.
- Hinreichendes Kriterium liefert folgende linear laufende Heuristik:
  1. Wähle Knoten n mit Grad kleiner k aus.
  2. Nicht möglich? Ausgabe: Weiß nicht, ob k-färbbar. ($\Rightarrow$ *Auslagern*)
  3. Sonst eliminiere n und seine Kanten.
  4. Gehe zu 1. wenn Graph nicht leer.
     Sonst Ausgabe: k-färbbar.
- Färbe Graphen in umgekehrter Eliminierungsfolge.

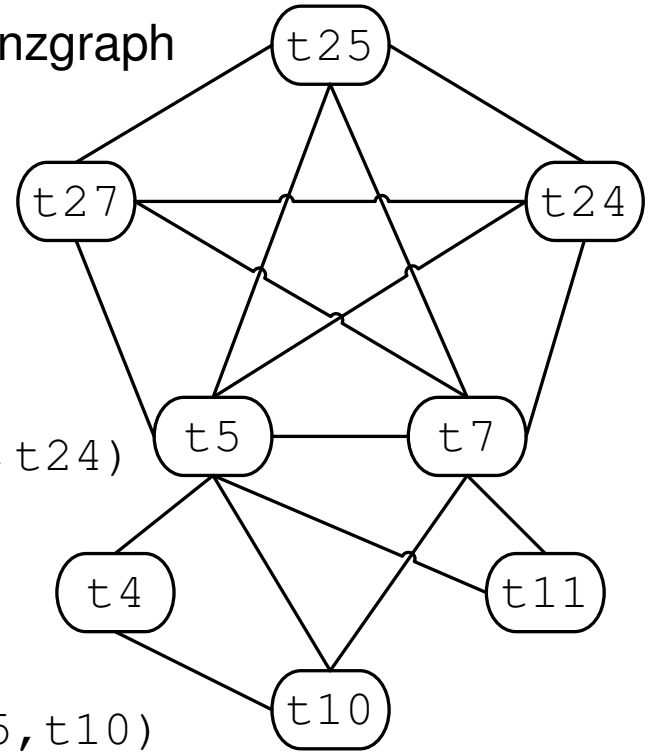
Beispiel – Interferenzgraph

Lebens-
zeiten

Programm

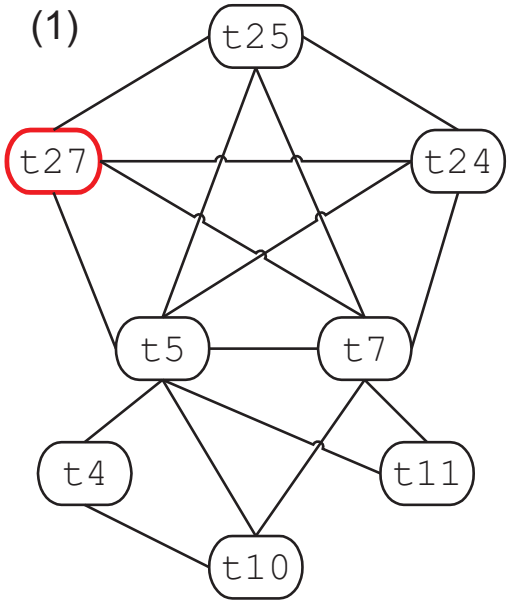
```
t5  = 1
t7  = 2
t27 = 3
t25 = 4
t24 = 5
call(&x,t5,t7,t27,t25,t24)
t11 = t5 + t7
st(t11, &y)
t10 = t5 + t7
st(t10, &z)
t4  = 0
call(&printf,“%x“,t4)
call(&printf,“%x%x“,t5,t10)
```

Interferenzgraph

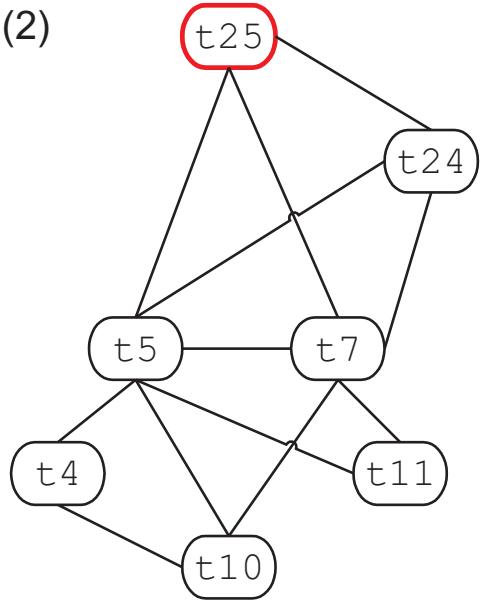


Beispiel – Knoten eliminieren ($k = 5$)

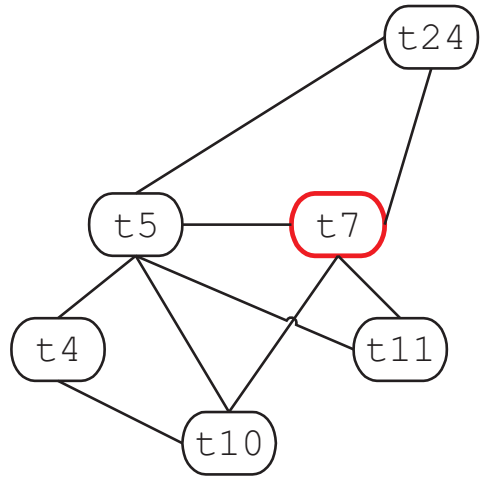
(1)



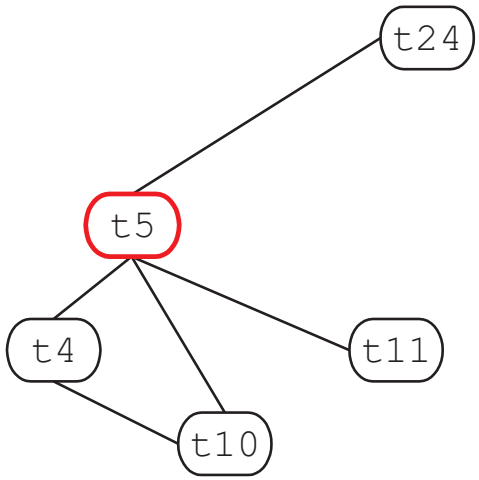
(2)



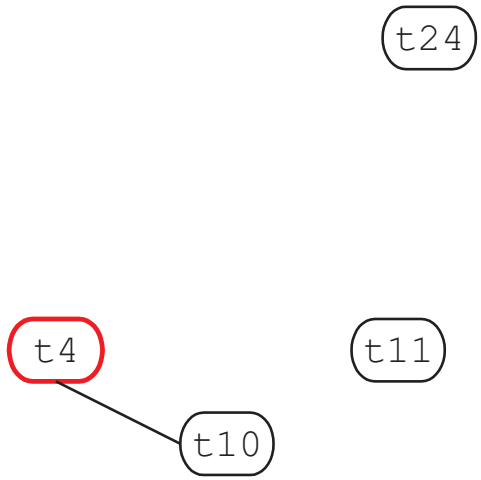
(3)



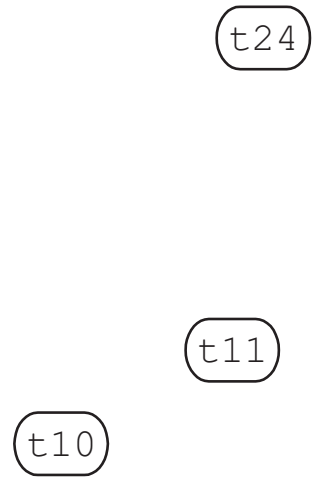
(4)



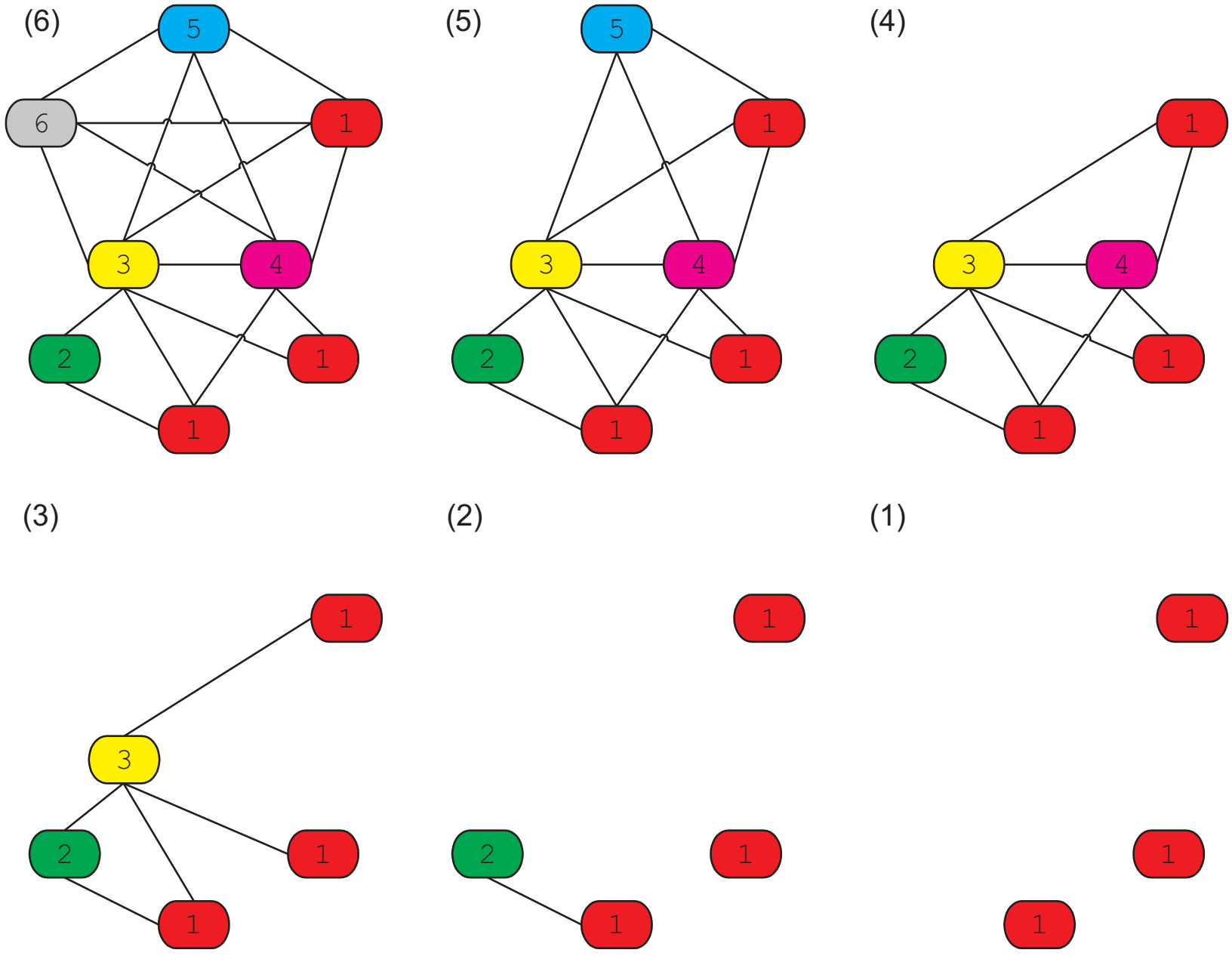
(5)



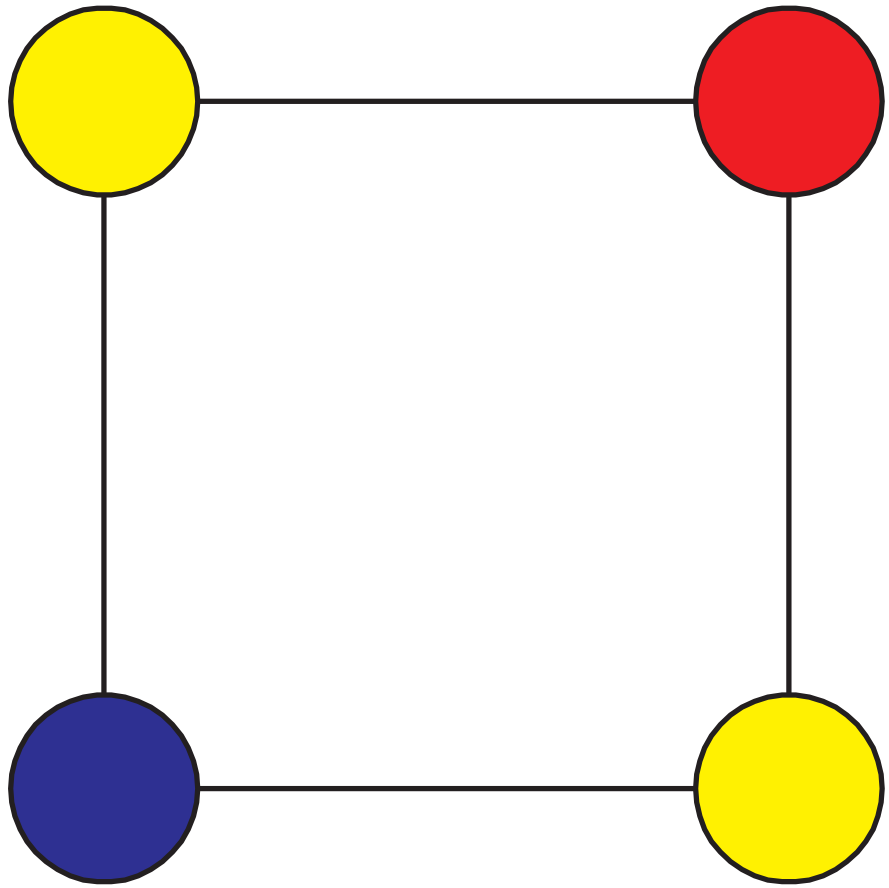
(6)



Beispiel – Färbung der Knoten



Offensichtlich 2-färbbar aber Heuristik scheitert

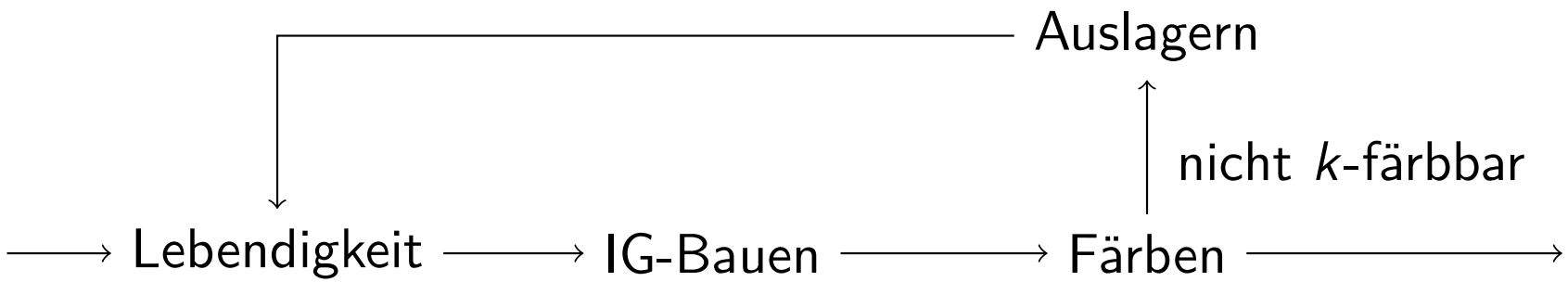


Heuristik unterstellt, daß alle Nachbarknoten unterschiedlich gefärbt sein müssen.

Färbung nicht gefunden

Wenn Färbung nicht gefunden wurde, kann die Heuristik iteriert werden:

- Eliminiere eine Knoten m aus Register-Interferenz-Graph
- Entsprechender Wert kommt nicht in globales Register, sondern wird in den Speicher ausgelagert.
- Neuer Versuch:
  Graphenfärben des Rest-Register-Interferenz-Graphen
- Auswahl von m heuristisch siehe 1. bis 4. der nachfolgenden Rangfolge



Register auslagern

- Genügen die Register nicht („Registerdruck zu hoch", Graph nicht k-färbbar), so müssen Werte in den Speicher ausgelagert werden
- Auswahl der auszulagernden Werte (Rangfolge):
  1. Wert kann mit einem (oder wenigen) Befehlen aus anderen Registerinhalten wieder berechnet werden
  2. Wert schon im Speicher vorhanden oder mit einem Speicherzugriff wiederberechenbar
  3. Wert wird möglichst lange nicht benötigt
  4. Wert interferiert mit vielen anderen
- Bei 1. und 2. kein Auslagern nötig, 3. nur angenähert beurteilbar, z.B. innerhalb eines Grundblocks
- Probleme:
  - Auslagern kann während der Registerzuteilung, aber auch danach nötig werden, z.B. während Befehlsanordnung
  - Befehlsanordnung kann die Bedingungen verändern

Weitere Verbesserungen

Bevor die Registerzuteilung beginnt (Chow & Hennessy):

- Konstanten aufspalten, d.h. die Konstanten die in zusammengezogen wurden (CSE), unmittelbar vor ihrer Verwendung in den Code platzieren.
- Allgemeiner – Rematerialisierung: Werte die sich leicht (wenige Takte $<$ Speicherzugriffzeit) wiederberechnen lassen, nicht in Register belassen

Verbesserung zu Chaitins Algorithmus (Briggs):

- Optimistisches Färben (Briggs): Eliminiere Knoten mit Grad k trotzdem; Auslagern erst falls Färben nicht möglich.
- Aufteilen der Lebenszeiten durch SSA-Form begünstigt Registerzuteilung.

Literatur I


Preston Briggs, Keith D. Cooper, and Linda Torczon.
Improvements to graph coloring register allocation.
ACM Transactions on Programming Languages and Systems, 16(3):428–455, May 1994.


G. J. Chaitin.
Register allocation & spilling via graph coloring.
In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, USA, 1982. ACM Press.


Sebastian Hack, Daniel Grund, and Gerhard Goos.
Register allocation for programs in SSA-form.
In *Compiler Construction*, volume 3923. Springer, March 2006.

Literatur II


Massimiliano Poletto and Vivek Sarkar.
Linear scan register allocation.
ACM Transactions on Programming Languages and Systems, 21(5):895–913, 1999.


Christian Wimmer and Hanspeter Mössenböck.
Optimized interval splitting in a linear scan register allocator.
In *VEE '05: Proceedings of the 1st international conference on Virtual execution environments*, pages 132–141, New York, NY, USA, 2005. ACM.