Kapitel 9
Registerzuteilung

Kapitel 9: 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 Register.
Prinzip: Weise den Variablen Register zu. Falls nicht möglich, lagere Werte in Hauptspeicher aus.

Registerklassen

Register können nicht beliebig verwendet werden:

- Gebrauch der Register durch Hardware festgelegt:
  - Gleitkomma-/Integer Register
  - Spezialregister: 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, Gleitkomma, Spezialregister (Rahmenzeiger, Bedingungsanzeige). Zuteilungsverfahren betrachten die Klassen separat.

Registerzuteilung: Wann/Wo

„Wann“:

- Nach Befehlsauswahl; Probleme:
  - Kosten in Befehlsauswahl von Registerzuteilung abhängig
  - Auslagerungscode (*spill-code*) von Registerzuteilung abhängig
  - Bestimmter Code nur mit bestimmten Registern auswählbar
- Vor Befehlsauswahl; Probleme:
  - Befehlsauswahl definiert Anzahl der benötigten Register
  - Manche Werte werden nie explizit berechnet, z.B. Werte auf Adressierungspfaden
- Befehlsauswahl – Registerzuteilung – Befehlsauswahl
- Während der Befehlsauswahl (*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 gleichem Quell- und Zielregister entfernen)

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

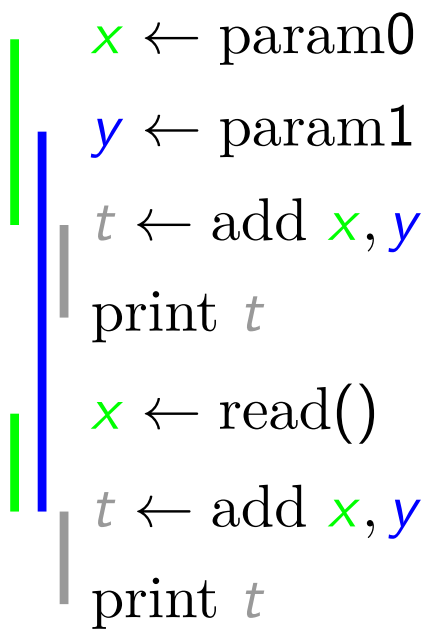
Kapitel 9: Registerzuteilung

Grundlagen: Lebendigkeit

Definition:

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

Beispiel: Lebendigkeit für x, y, t



Berechnung Lebendigkeit, 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).

Definition:

2 Variablen interferieren wenn sie an einem Programmpunkt beide lebendig sind.

Interferieren 2 Variablen so müssen sie unterschiedliche Registern zugeteilt bekommen!

Grundlagen: Auslagern

Was tun wenn Register nicht ausreichen? In den Speicher (Activation Record) auslagern.

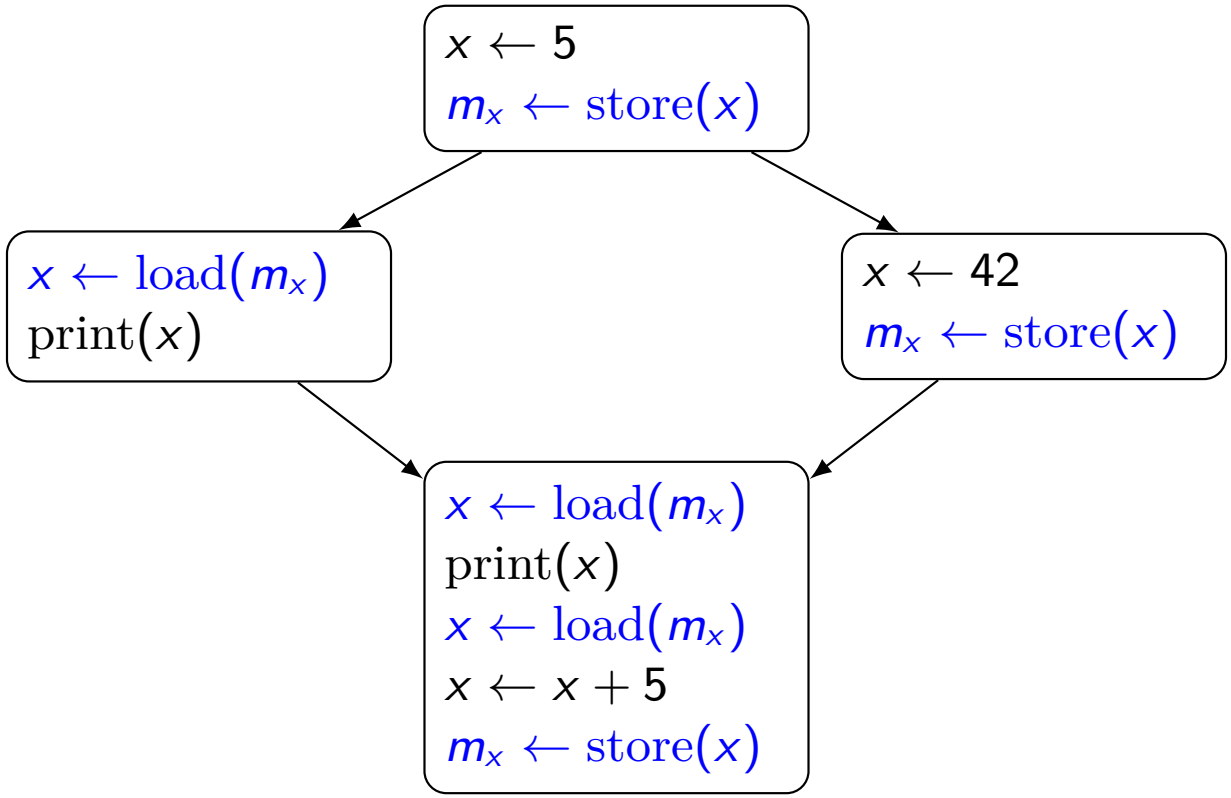
Übliches Verfahren:

- Nach jeder Definition Wert in Activation Record sichern
- Vor jeder Verwendung Wert aus Activation Record laden
- $\Rightarrow$ Lebenszeit minimal

Welche Variable zum Auslagern wählen? Kostenmaß:

1. Wert kann einfach wieder berechnet werden (Konstanten, Parameter)
2. Wert wird lange nicht benötigt
3. Wert interferiert mit vielen anderen
4. ...

Auslagern – Beispiel



- Gute Platzierung der Speicher- und Ladebefehle
  NP-vollständig $\Rightarrow$ Forschungsthema

Verfahren zur Zuteilung von Registern

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

Kapitel 9: Registerzuteilung

Lokale Registerzuteilung mit Freiliste (*on the fly*)

- Bestimme letzte Verwendungen von Werten im Grundblock.
- Hilfsfunktionen:
  - *allocReg*(): Gibt ein freies Register zurück, falls keines mehr frei ist, löse Ausnahme aus. Entferne Register aus der Freiliste.
  - *freeReg*(r): Füge Register r in die Freiliste ein.
- Durchlaufe Grundblock von Anfang bis Ende:
  - Falls Register benötigt wird rufe *allocReg*() auf
  - Nach letzter Verwendung rufe *freeReg*(r).
- 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 Compiler funktionierten wirklich so), ggf. muss dieses Verfahren um die Möglichkeit des Auslagerns erweitert werden.

Lokale Registerzuteilung – Beispiel

b3 ← xor b3, b2
b4 ← b2
b1 ← and b1, b3
b1 ← xor b4, b2
b4 ← and b3, b2
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
jmp otherBlock

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste:
Allokation:

Lokale Registerzuteilung – Beispiel

b3 ← xor b3, b2
b4 ← b2
b1 ← and b1, b3
b1 ← xor b4, b2
b4 ← and b3, b2
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
jmp otherBlock

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste: $\{R0, R1, R2, R3, R4\}$
Allokation:

Lokale Registerzuteilung – Beispiel

R0 ← load %fp + $b3
R1 ← load %fp + $b2
b3 ← xor b3, b2
b4 ← b2
b1 ← and b1, b3
b1 ← xor b4, b2
b4 ← and b3, b2
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
jmp otherBlock

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste: $\{R2, R3, R4\}$
Allokation:

- b3 – R0
- b2 – R1

Lokale Registerzuteilung – Beispiel

R0 ← load %fp + $b3
R1 ← load %fp + $b2
b3 ← xor b3, b2
b4 ← b2
b1 ← and b1, b3
b1 ← xor b4, b2
b4 ← and b3, b2
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
jmp otherBlock

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste: $\{R3, R4\}$
Allokation:

- b3 – R0
- b2 – R1
- b4 – R2

Lokale Registerzuteilung – Beispiel

R0 ← load %fp + $b3
R1 ← load %fp + $b0
b3 ← xor b3, b2
b4 ← b2
R3 ← load %fp + $b1
b1 ← and b1, b3
b1 ← xor b4, b2
b4 ← and b3, b2
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
jmp otherBlock

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste: $\{R4\}$
Allokation:

- b3 – R0
- b2 – R1
- b4 – R2
- b1 – R3

Lokale Registerzuteilung – Beispiel

R0 ← load %fp + $b3
R1 ← load %fp + $b0
b3 ← xor b3, b2
b4 ← b2
R3 ← load %fp + $b1
b1 ← and b1, b3
b4 ← xor b4, b2
b4 ← and b3, b2
store R1, %fp + $b2
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
jmp otherBlock

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste: $\{R1, R4\}$
Allokation:

- b3 – R0
- b2 – R1 (geschrieben)
- b4 – R2
- b1 – R3

Lokale Registerzuteilung – Beispiel

R0 ← load %fp + $b3
R1 ← load %fp + $b2
b3 ← xor b3, b2
b4 ← b2
R3 ← load %fp + $b1
b1 ← and b1, b3
b1 ← xor b4, b2
b4 ← and b3, b2
store R1, %fp + $b2
R1 ← load %fp + $b0
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
jmp otherBlock

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste: $\{R1, R4\}$
Allokation:

- b3 – R0
- b2 – R1 (geschrieben)
- b4 – R2
- b1 – R3
- b0 – R1

Lokale Registerzuteilung – Beispiel

```
R0 ← load %fp + $b3
R1 ← load %fp + $b2
b3 ← xor b3, b2
b4 ← b2
R3 ← load %fp + $b1
b1 ← and b1, b3
b4 ← xor b4, b2
b4 ← and b3, b2
store R1, %fp + $b2
R1 ← load %fp + $b0
b0 ← or b0, b3
b1 ← xor b1, b4
b1 ← xor b1, b3
b3 ← or b3, b0
store R0, %fp + $b3
store R1, %fp + $b0
store R2, %fp + $b4
store R3, %fp + $b1
jmp otherBlock
```

1. Letzte Verwendungen bestimmen.
2. Grundblock durchlaufen.
3. Werte zurückschreiben.

Freiliste: $\{R0, R1, R2, R3, R4\}$
Allokation:

- b3 – R0
- b2 – R1 (geschrieben)
- b4 – R2
- b1 – R3
- b0 – R1

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 Befehlsauswahl kombiniert werden.

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

Kapitel 9: Registerzuteilung

Linear Scan Register Allokation

- 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.

Wiederholung: Grundblöcke

Grundblock: Befehlsfolge maximaler Länge mit Eigenschaft: wenn ein Befehl ausgeführt wird, dann alle genau einmal, also:

- Grundblock beginnt mit einer Sprungmarke,
- enthält keine weiteren Sprungmarken
- endet mit (bedingten) Sprüngen, enthält sonst keine weiteren Sprünge
- Unterprogrammaufrufe zählen nicht als Sprünge!

Ablaufgraph

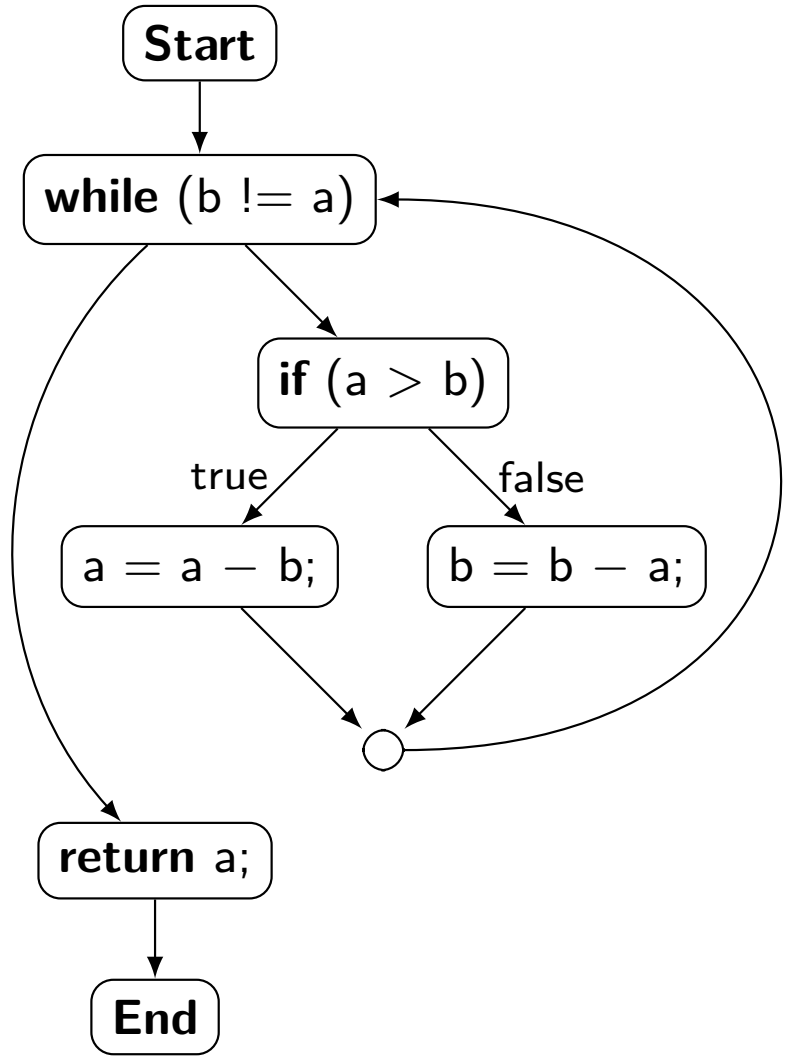
Ablaufgraph (engl. Control Flow Graph):

- Enthält einen Knoten für jeden Grundblock.
- Kanten zwischen Knoten repräsentieren mögliche Sprünge zwischen Grundblöcken.
- Es existieren 2 zusätzliche Knoten **Start** und **End**
- Es gibt eine Kante von **Start** zu jedem Grundblock mit dem das Programm betreten werden kann.
- Es gibt eine Kante von jedem Grundblock mit dem das Programm verlassen werden kann zu **End**.

Der Ablaufgraph ist eine *konservative Approximation*: Alle möglichen Programmabläufe sind enthalten, kann allerdings zusätzliche Kanten enthalten.

Beispiel Ablaufgraph

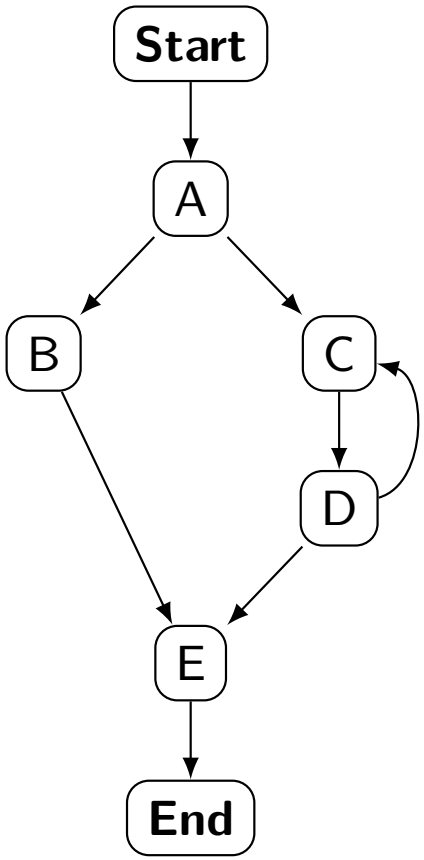
```
int gcd(int a, int b)
{
    while(b != a) {
        if(a > b)
            a = a − b;
        else
            b = b − a;
    }
    return a;
}
```



Grundlagen: Graphtraversierung

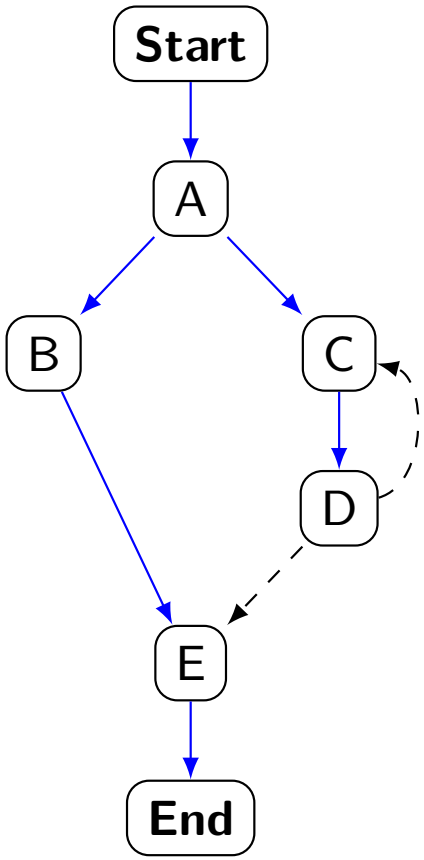
- Tiefensuche auf (Kontroll-)Flussgraph ergibt Tiefensuchbaum
- Baumtraversierungsverfahren damit auf Graphen erweitert:
  - Präfixordnung Betrachte erst Knoten, dann seine Kinder (im Tiefensuchbaum)
  - Postfixordnung Betrachte erst Kinder dann den Knoten
- Spezialfall: Umgekehrte Postfixordnung erzeugt für DAGs eine topologische Sortierung (vor jedem Kind kommen alle Eltern)
- Bei Ablaufgraphen gilt das auch, bis auf Schleifenköpfe

Beispiel Tiefensuchbaum



- Präfixordnung:
  Start, A, B, E, **End**, C, D
- Postfixordnung:
  End, E, B, D, C, A, **Start**
- Umgekehrte Postfixordnung:
  Start, A, C, D, B, E, **End**

Beispiel Tiefensuchbaum



- Präfixordnung:
  Start, A, B, E, **End**, C, D
- Postfixordnung:
  End, E, B, D, C, A, **Start**
- Umgekehrte Postfixordnung:
  Start, A, C, D, B, E, **End**

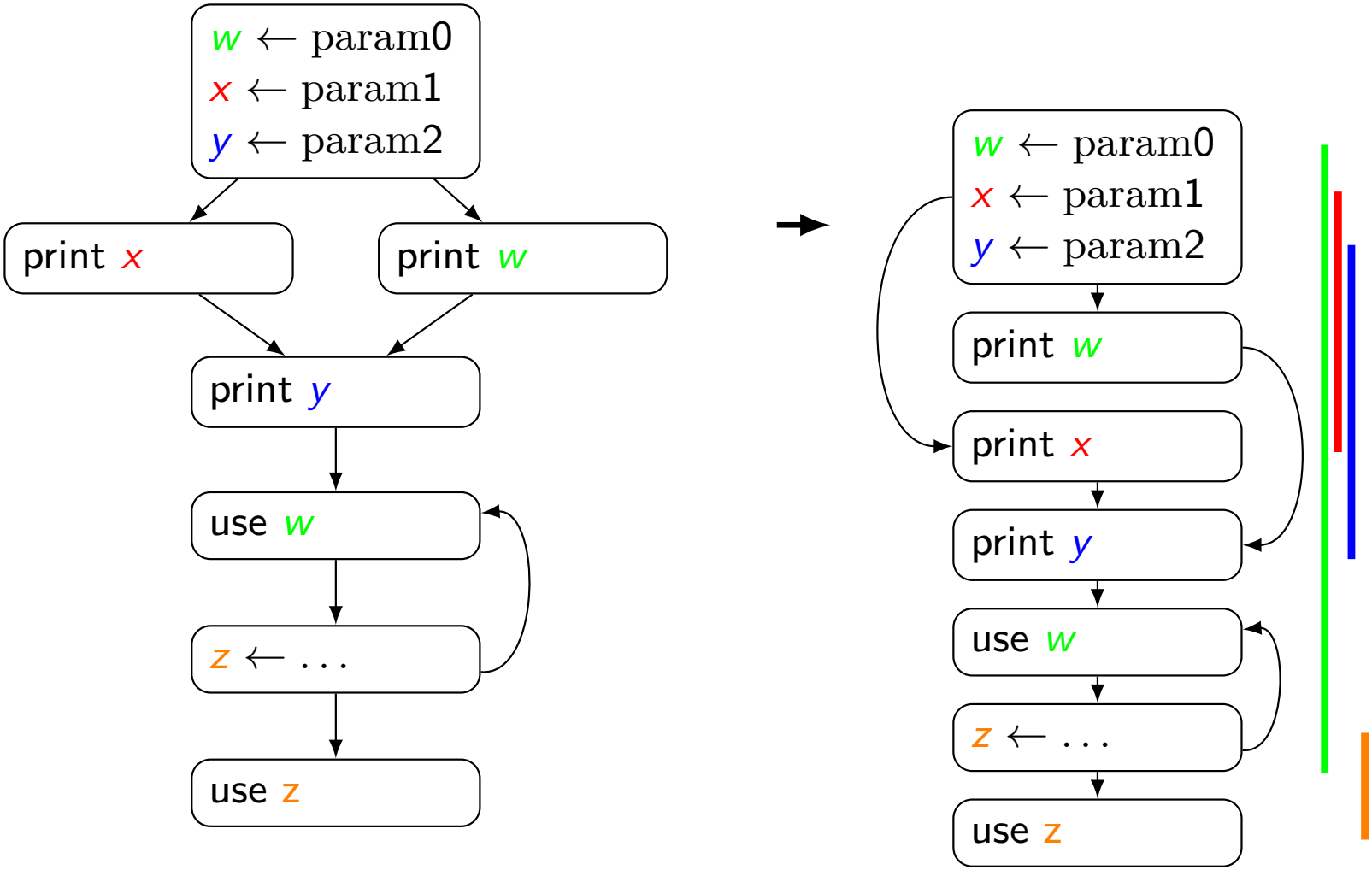
Tiefensuchbaum: Knoten + blaue Kanten
Umgekehrte Postfixordnung alle Vorgänger bevor Knoten betreten wird (abgesehen von Rückwärtskanten)

Linear Scan Algorithmus

Algorithmus

- Berechne (globale) Lebendigkeitsinformation
- Erstelle Liste von Grundblöcken in umgekehrter Postfixordnung
  $\Rightarrow$ Position von Befehlen als Zahl darstellbar
- Erzeuge Lebendigkeitsinterval für jede Variable. Interval enthält alle Punkte an denen Variable lebendig ist
- Durchlaufe sortierte Intervalliste:
  - Verwende *allocReg*() und *freeReg*() wie beim *on the fly* Ansatz
  - Auslagern bei Bedarf. Lagere längstes verbleibende Intervall zuerst aus

Linear Scan – Linearisierung



Eine Linearisierung kann Grundblöcke unnötigerweise überdecken.

Linear Scan – Erweiterungen

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

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

Kapitel 9: 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 falls Variablen interferieren
- Graphfärbung mit minimaler Farbanzahl (*chromatische Zahl* $\chi(G)$) liefert die Minimalanzahl benötigter Register und gleichzeitig die Registerzuteilung.

Algorithmus nach Chaitin

- Ist Register-Interferenz-Graph mit k (=Anzahl der Register) Farben färbbar?
  Aber: Bestimmung der chromatischen Zahl ist *NP-vollständig*
- Hinreichendes Kriterium liefert folgende lineare 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

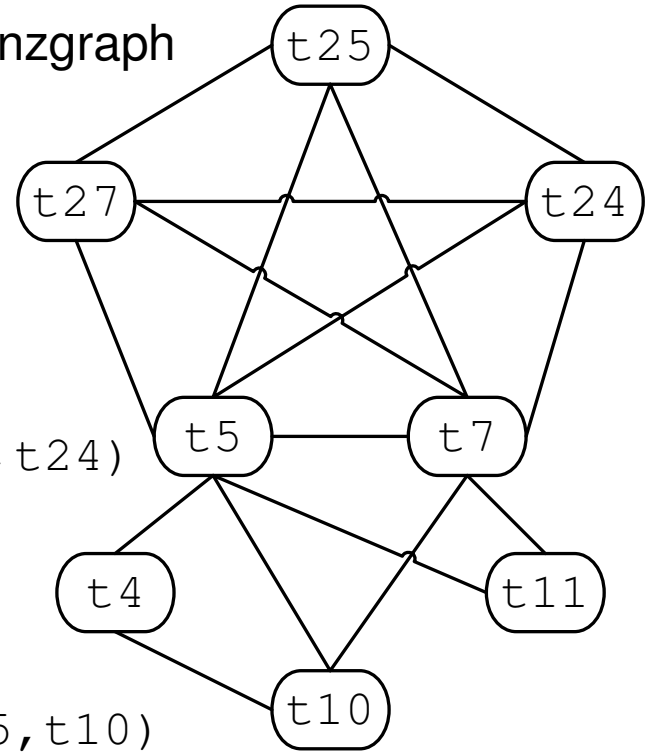
Beispiel – Interferenzgraph

Lebens-
zeiten

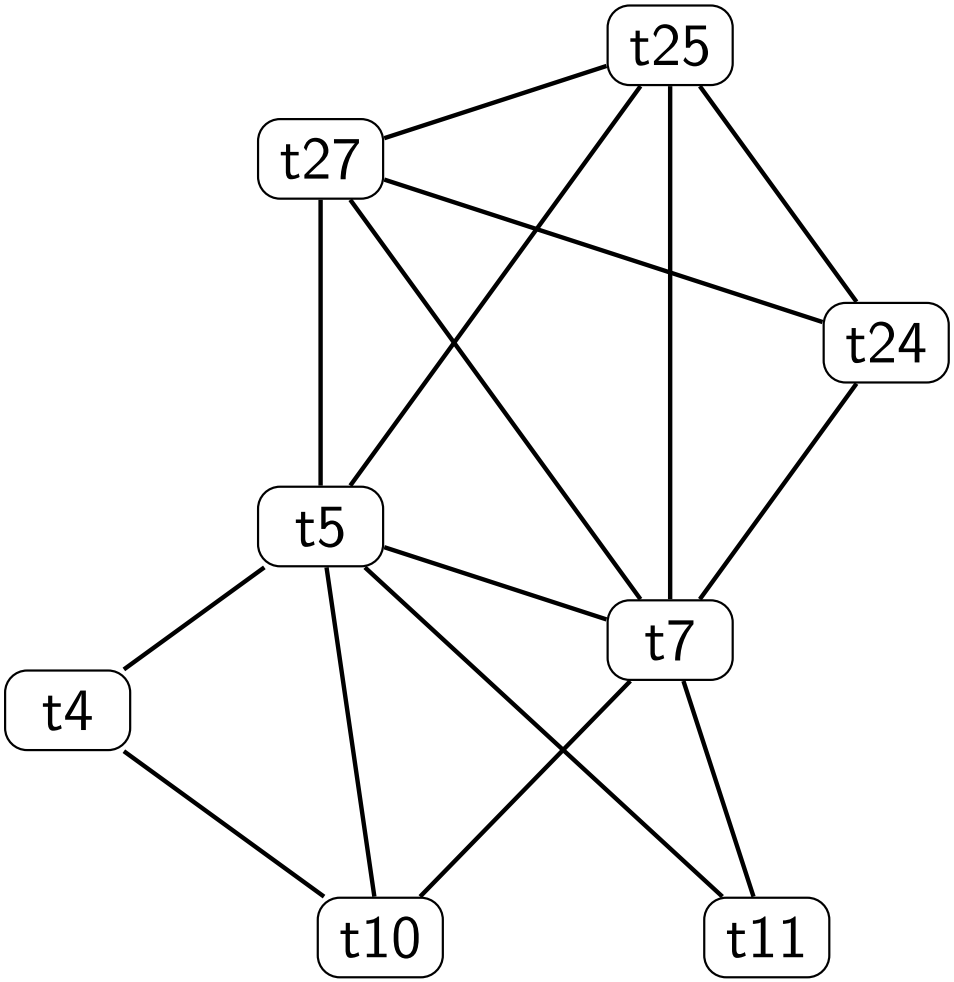
Programm

```
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



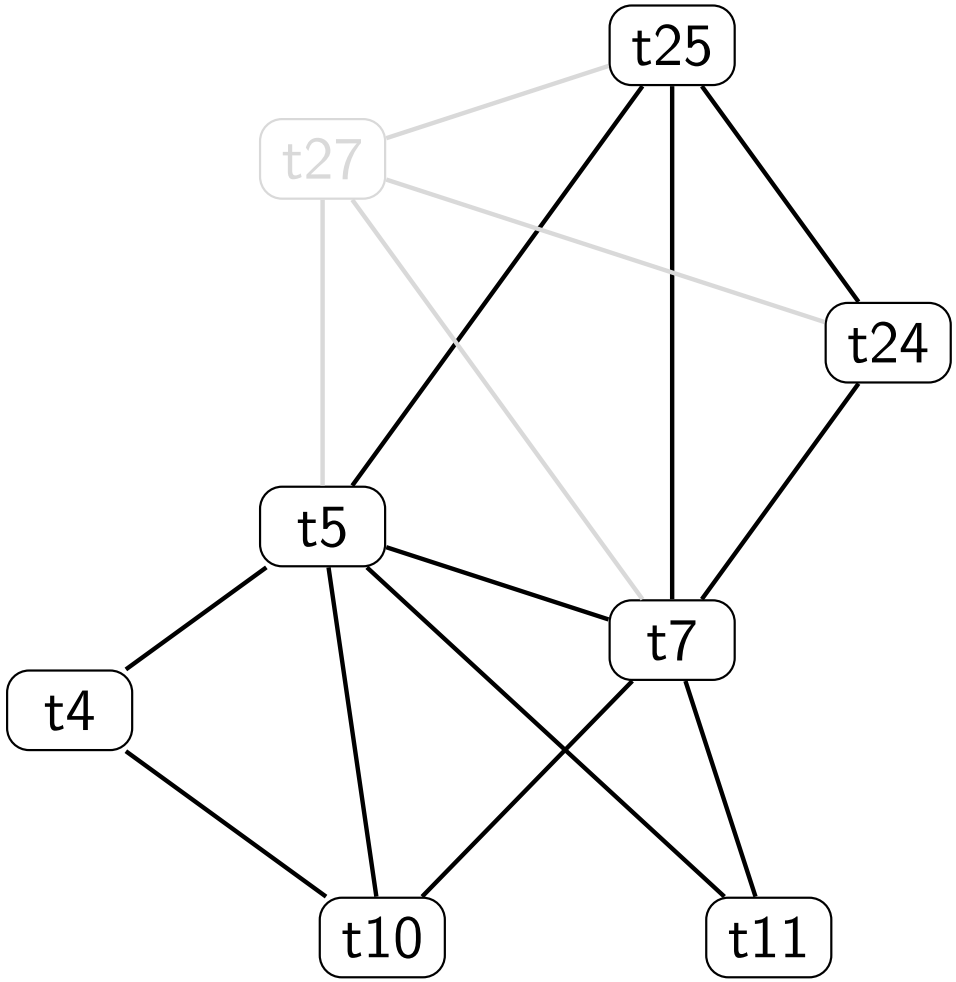
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

Eliminiert:

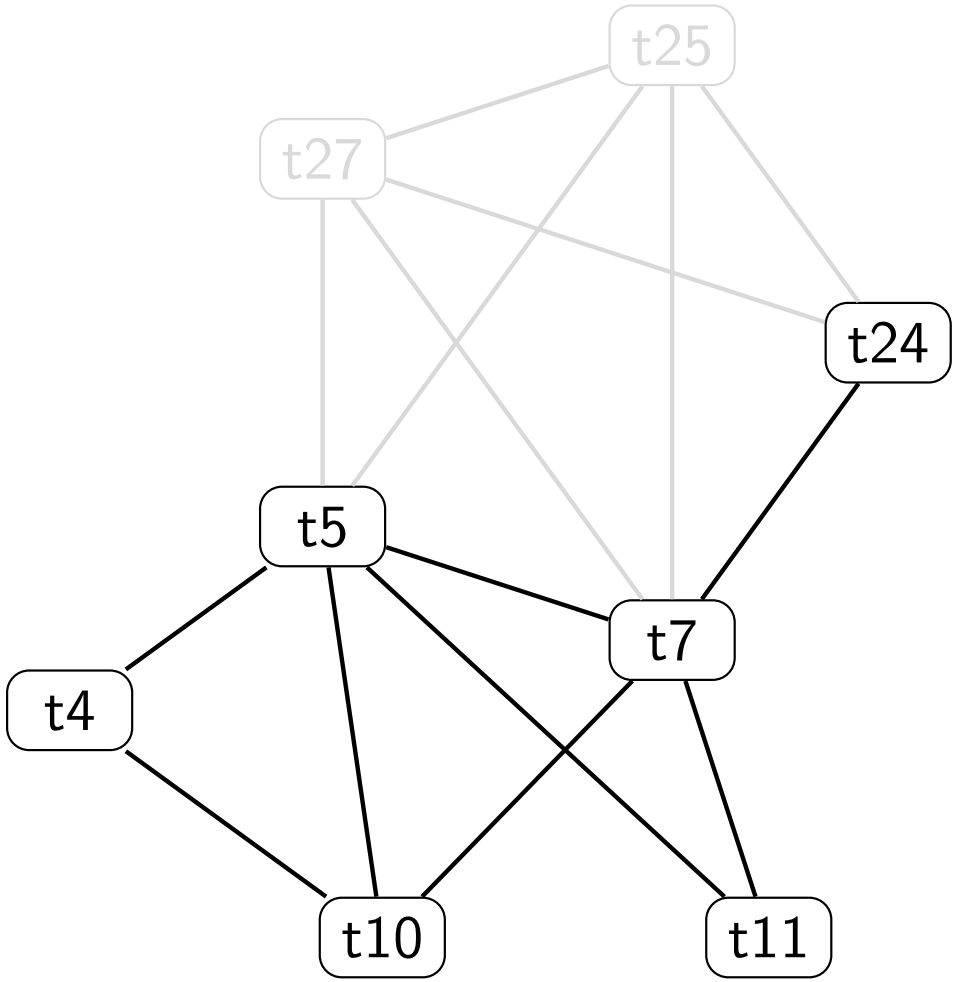
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

Eliminiert: t27,

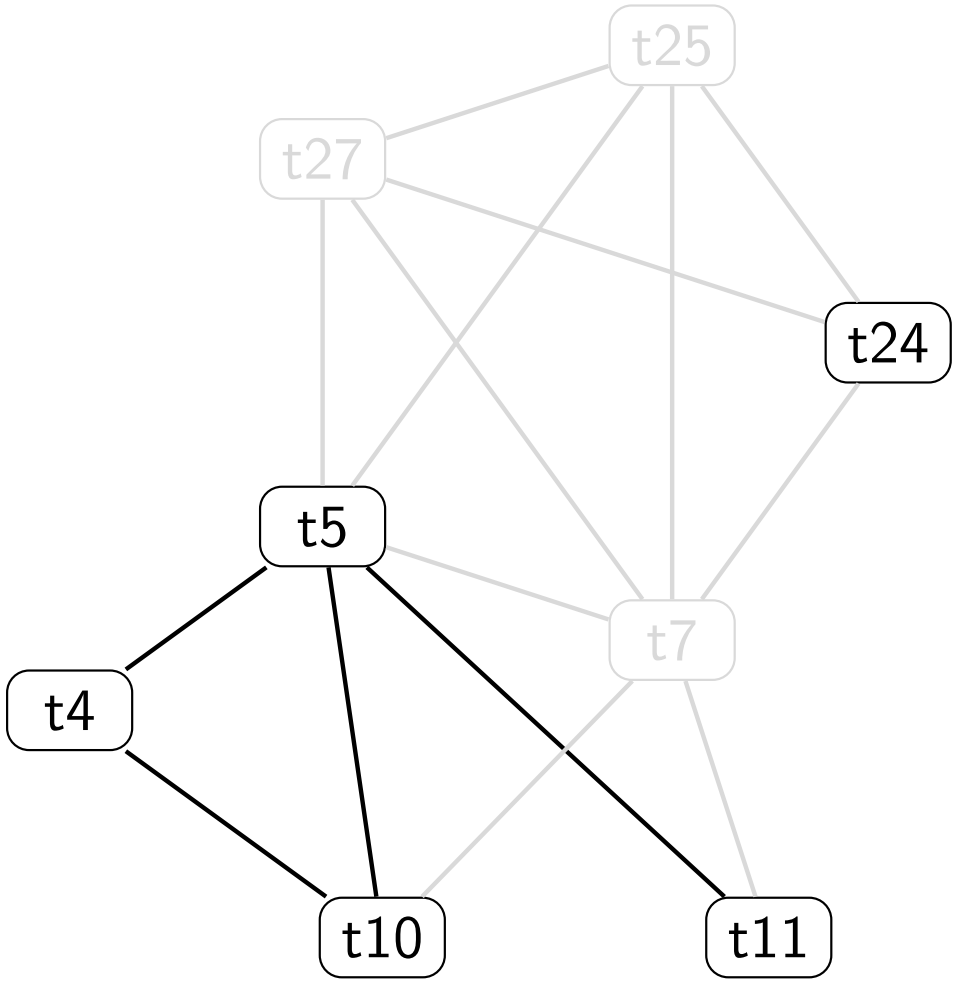
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

Eliminiert: t27, t25,

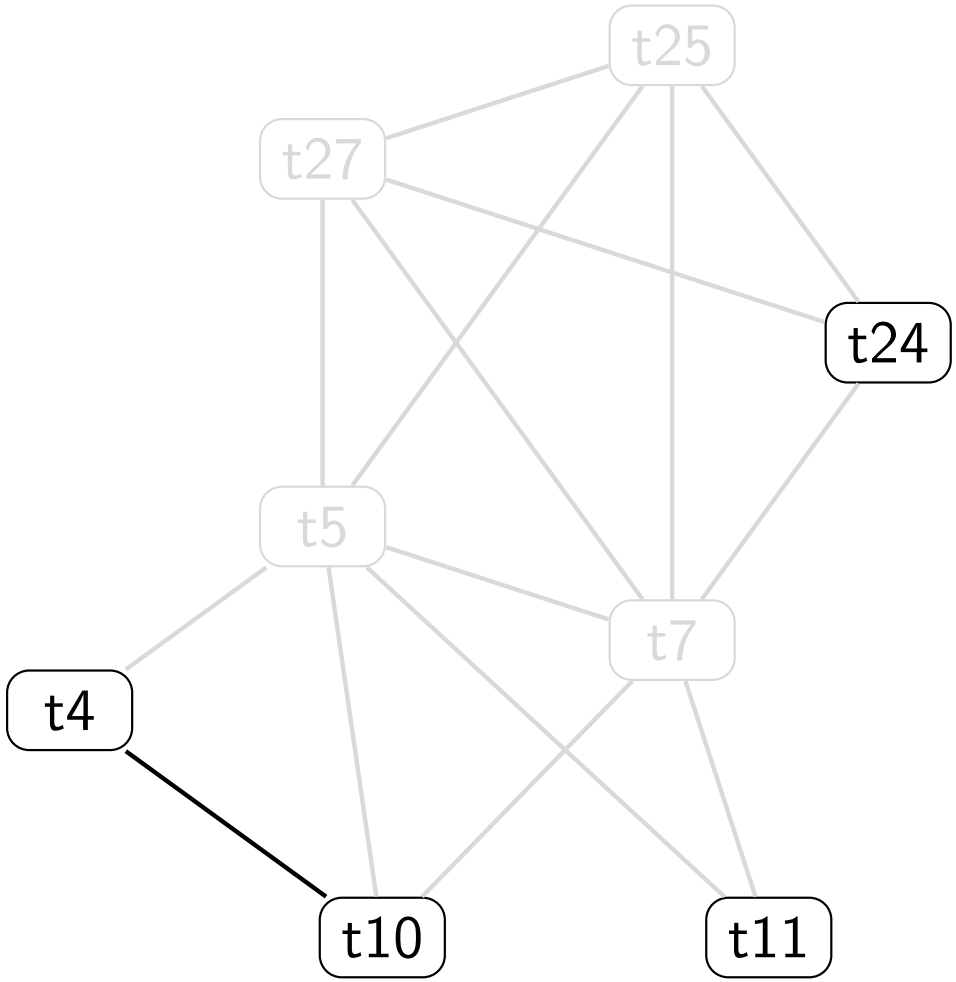
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

Eliminiert: t27, t25, t7,

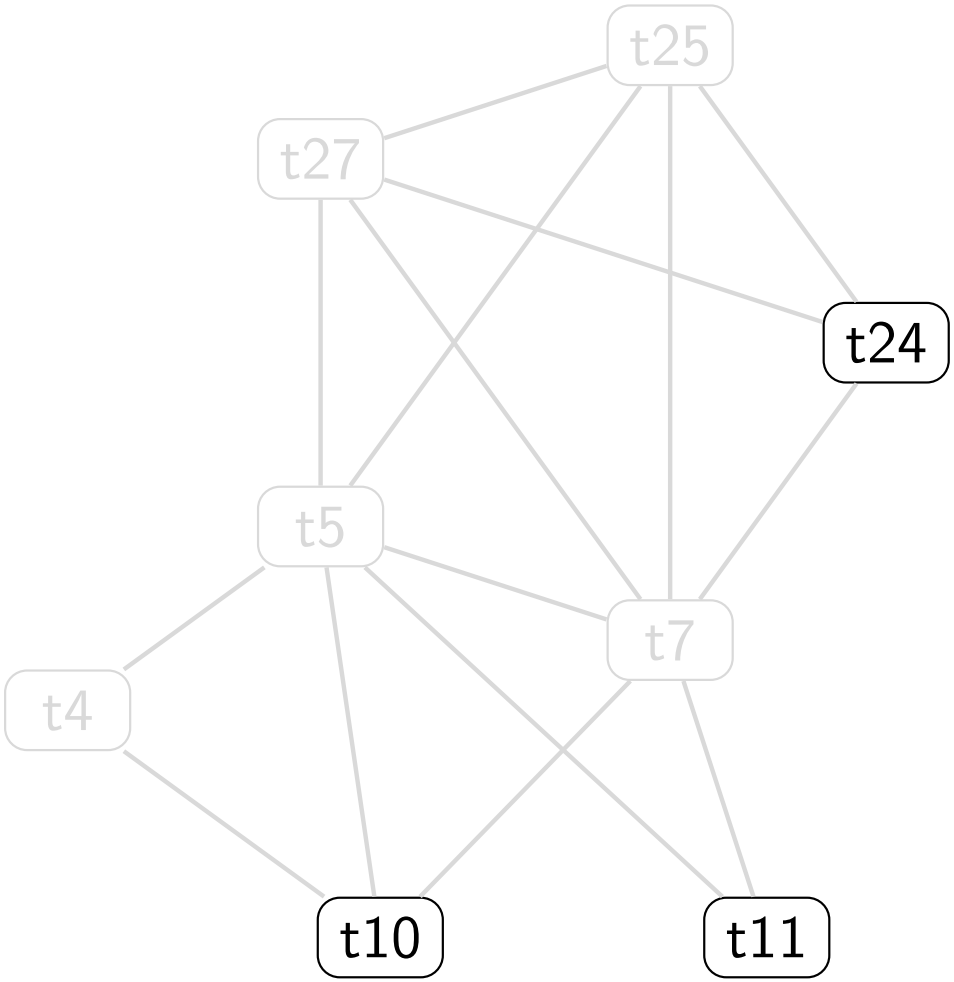
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

Eliminiert: t27, t25, t7, t5,

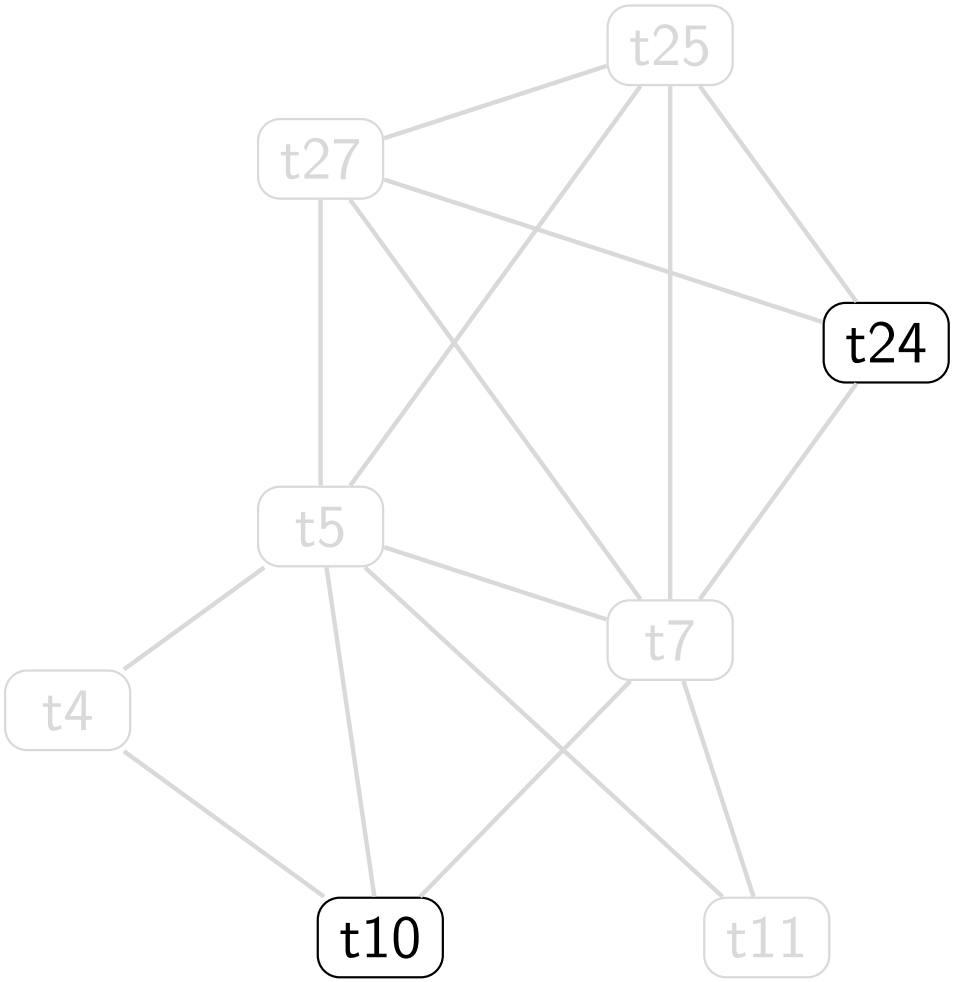
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

Eliminiert: t27, t25, t7, t5, t4,

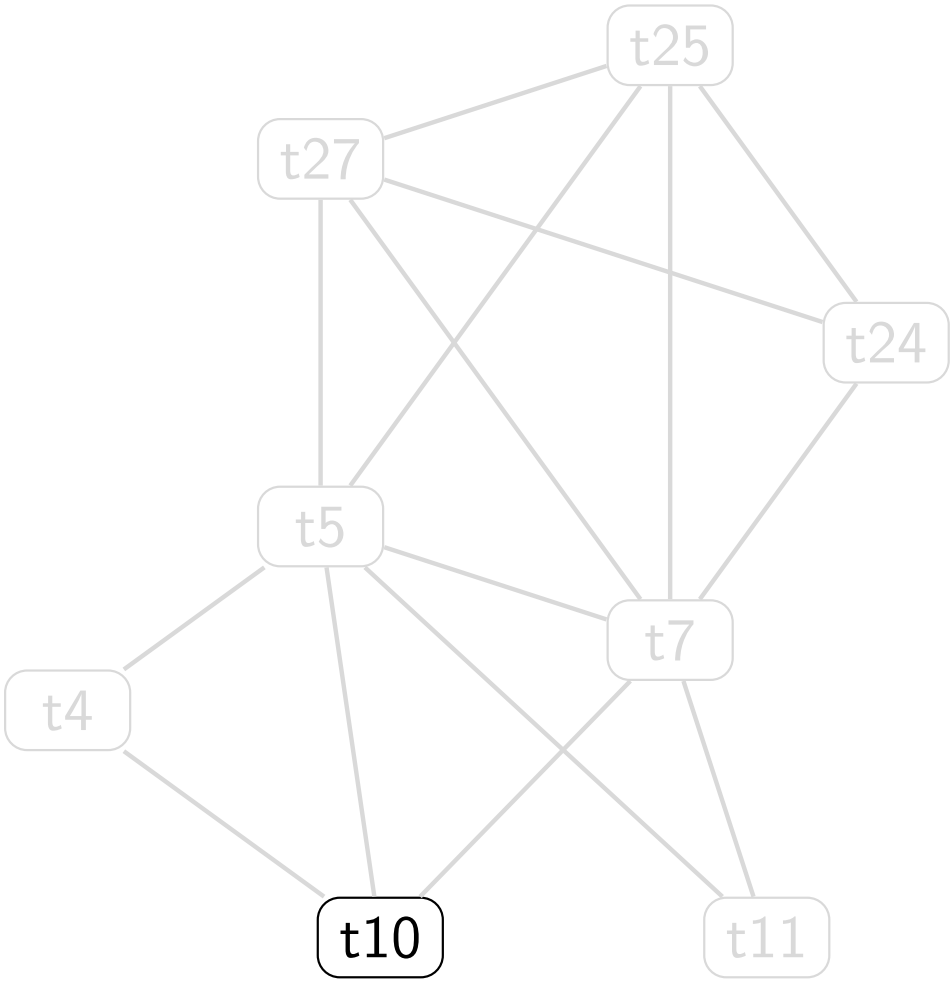
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

Eliminiert: t27, t25, t7, t5, t4, t11,

Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

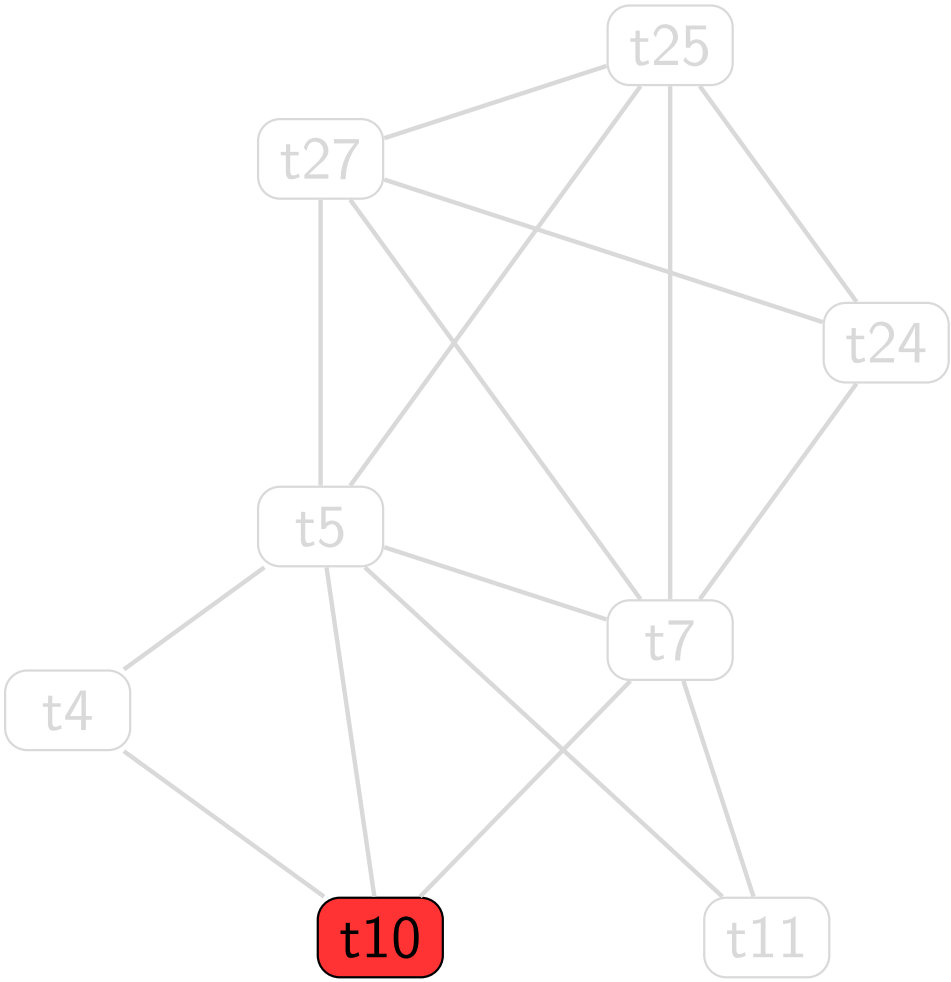
Eliminiert: t27, t25, t7, t5, t4, t11, t24,

Beispiel: Knoten eliminieren und färben ($k = 5$)

Registerallokation:

Eliminiert: t27, t25, t7, t5, t4, t11, t24, t10

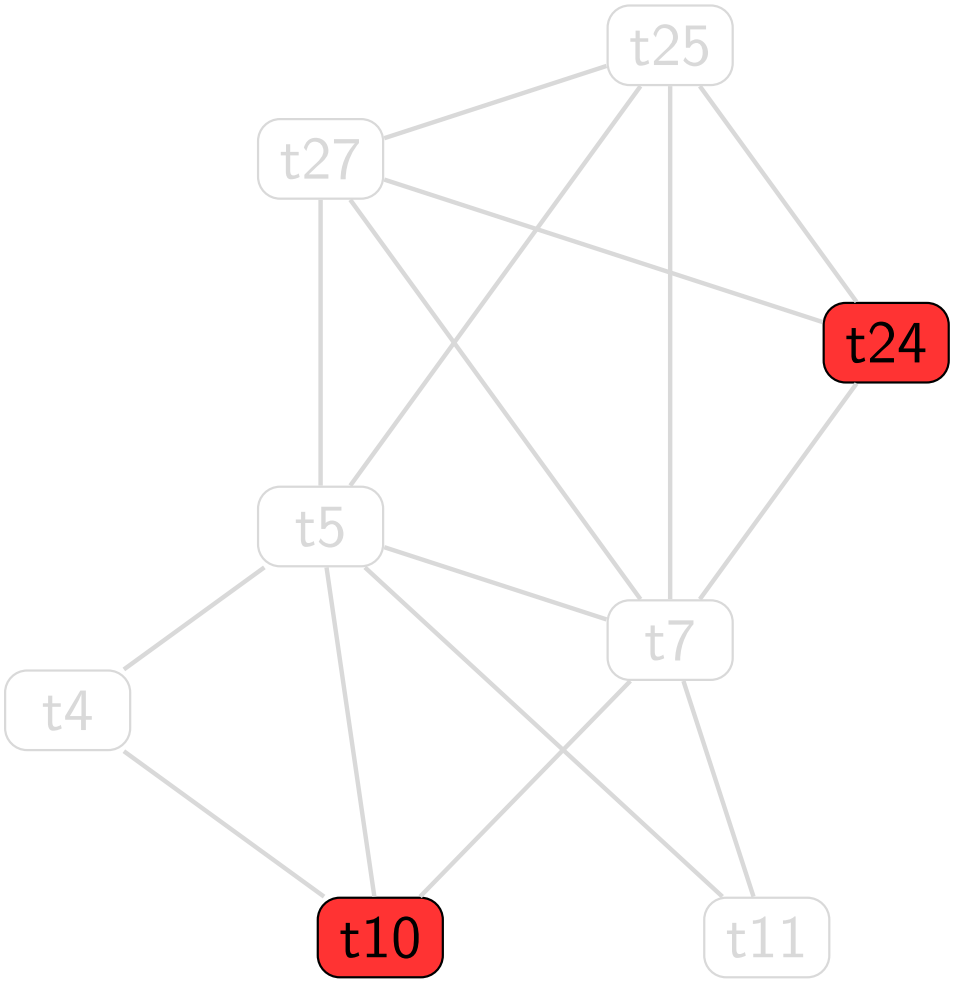
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:
t10: R1

Eliminiert: t27, t25, t7, t5, t4, t11, t24,

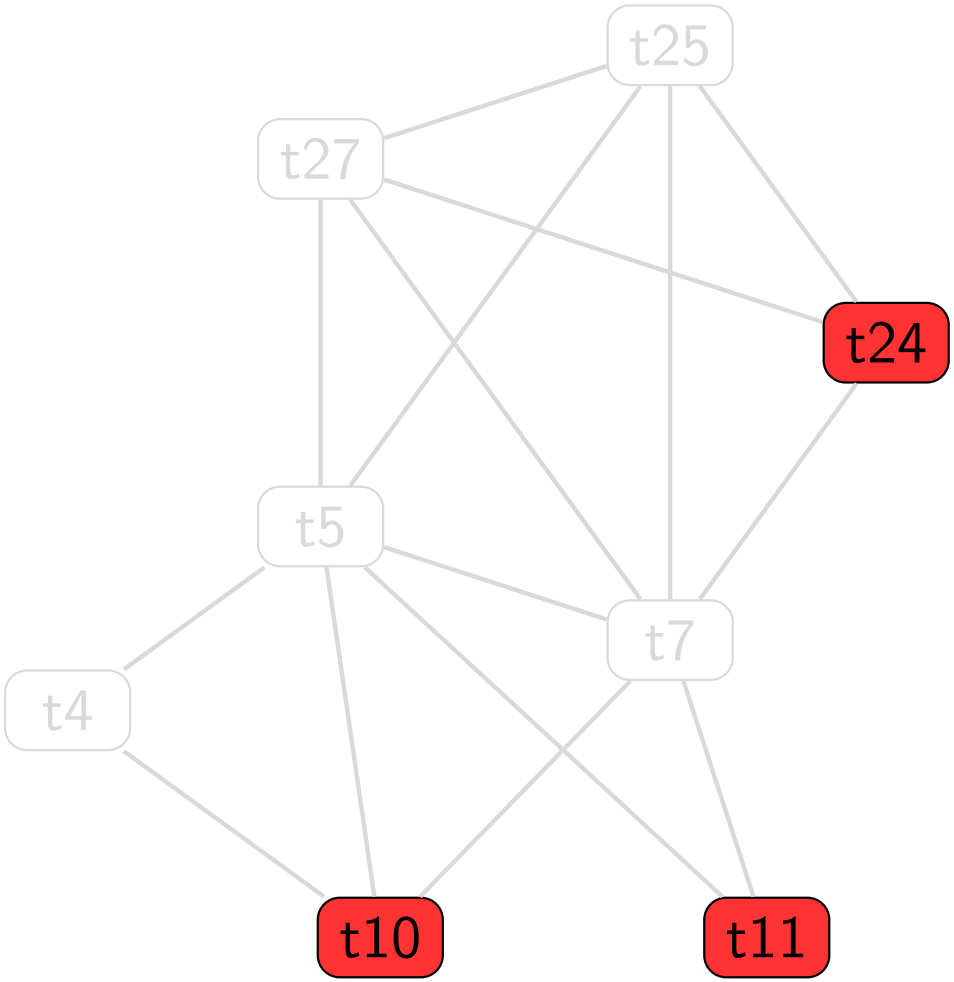
Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:
t10: R1
t24: R1

Eliminiert: t27, t25, t7, t5, t4, t11,

Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

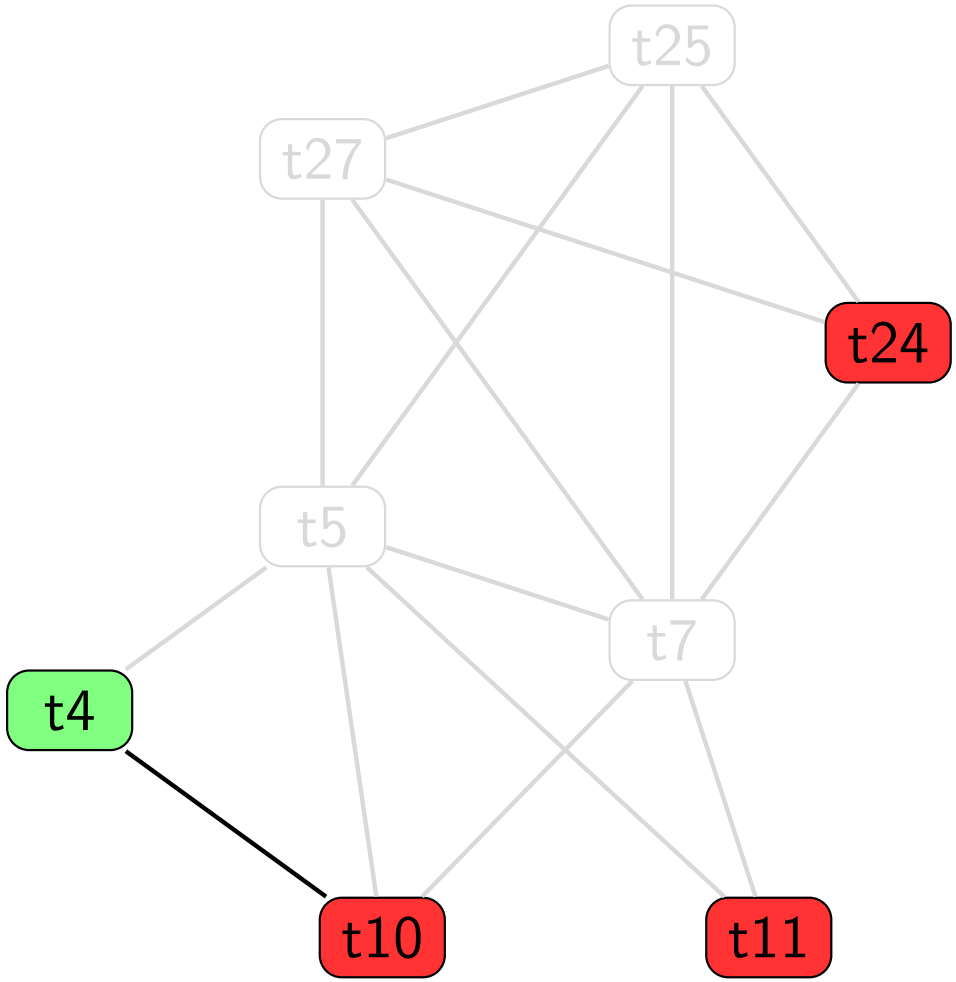
t10: R1

t24: R1

t11: R1

Eliminiert: t27, t25, t7, t5, t4,

Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

t10: R1

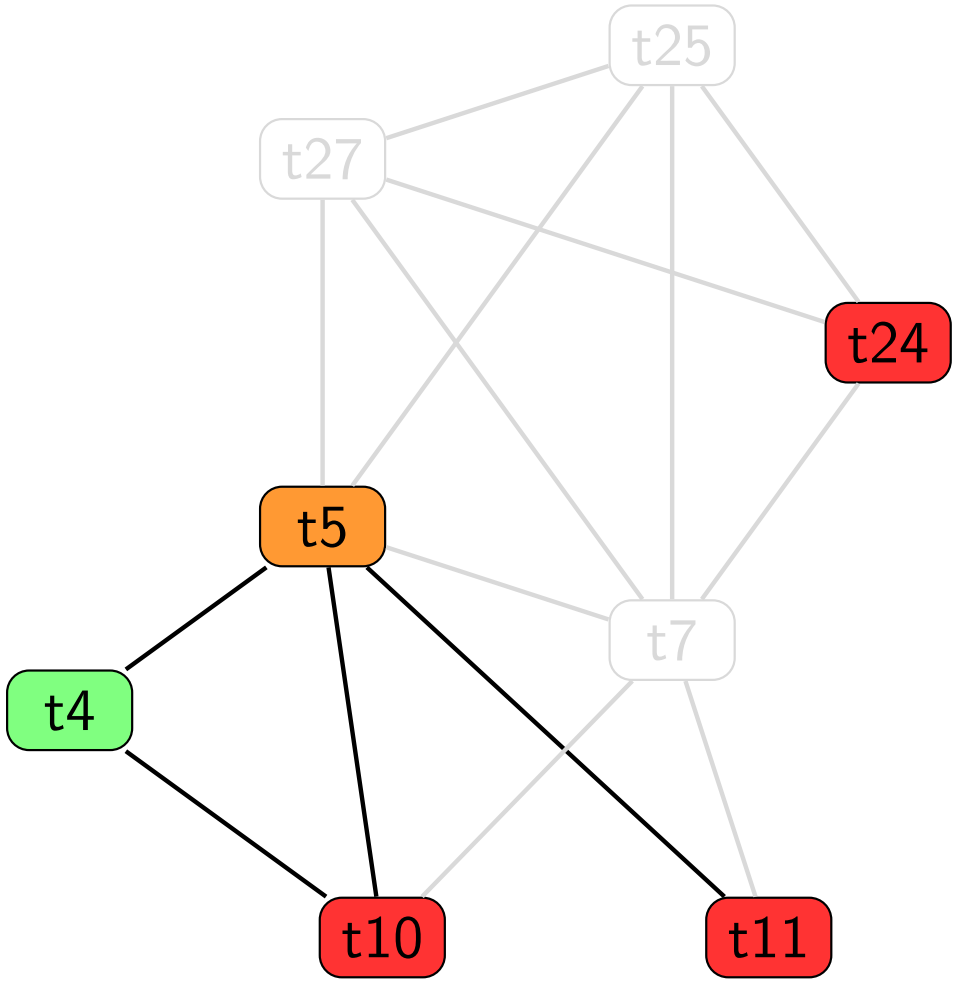
t24: R1

t11: R1

t4: R2

Eliminiert: t27, t25, t7, t5,

Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

t10: R1

t24: R1

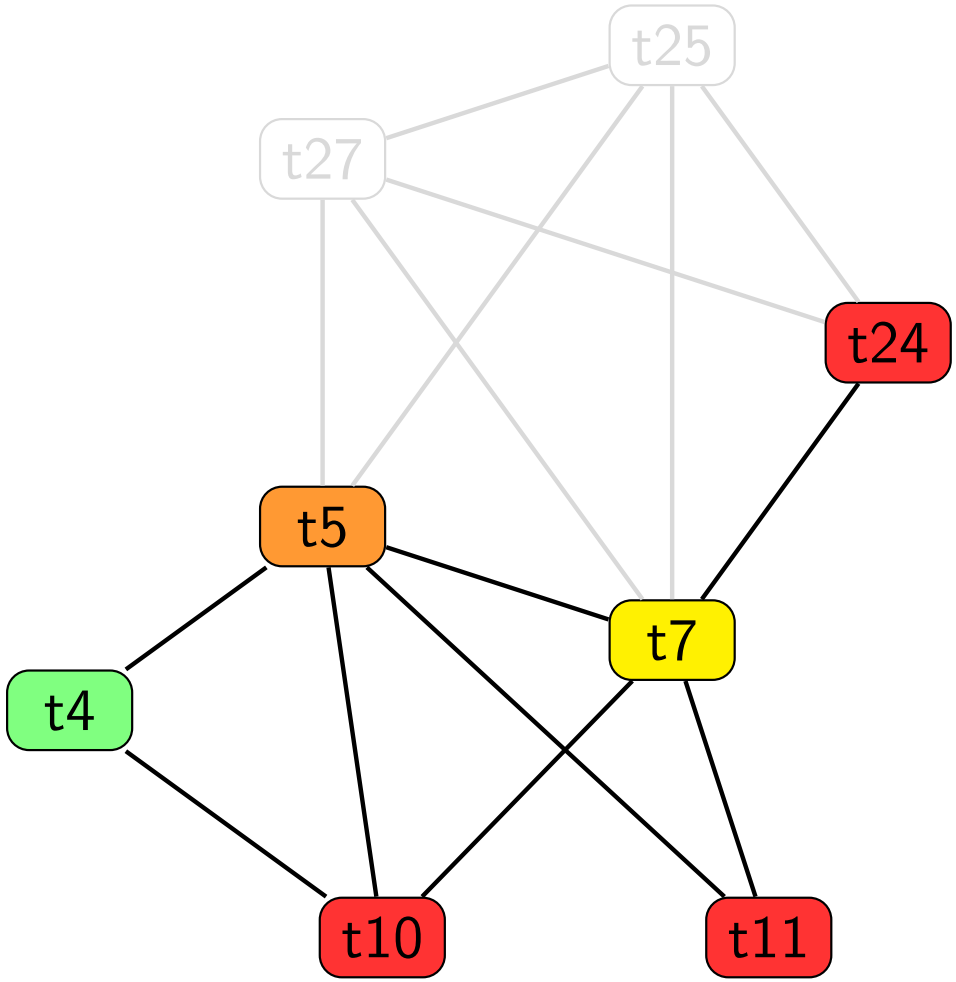
t11: R1

t4: R2

t5: R3

Eliminiert: t27, t25, t7,

Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

t10: R1

t24: R1

t11: R1

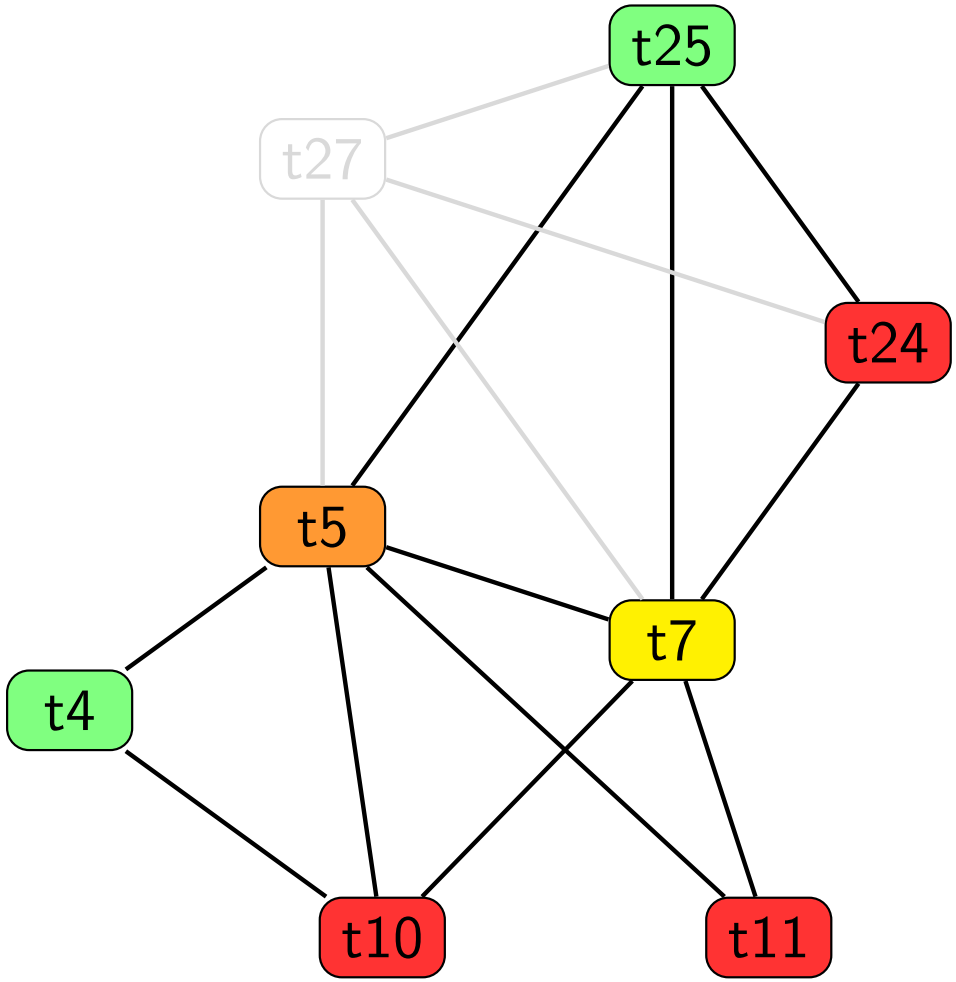
t4: R2

t5: R3

t7: R4

Eliminiert: t27, t25,

Beispiel: Knoten eliminieren und färben ($k = 5$)

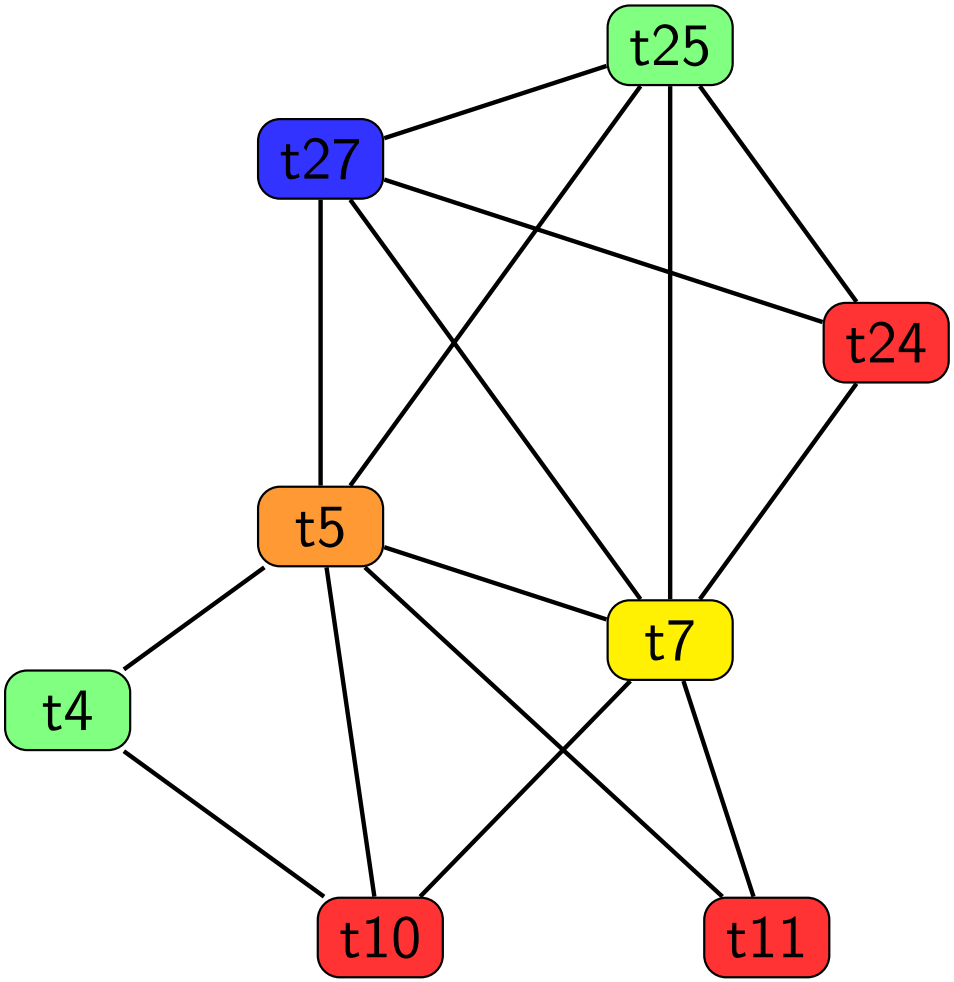


Registerallokation:

t10: R1
t24: R1
t11: R1
t4: R2
t5: R3
t7: R4
t25: R2

Eliminiert: t27,

Beispiel: Knoten eliminieren und färben ($k = 5$)



Registerallokation:

t10: R1
t24: R1
t11: R1
t4: R2
t5: R3
t7: R4
t25: R2
t27: R5

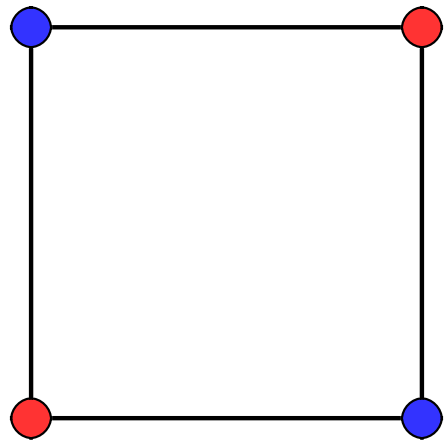
Eliminiert:

Iteratives Auslagern/Färben

Was, wenn kein Knoten mit Grad $< k$ existiert?

1. Entferne Knoten aus Interferenz-Graph bis ein Knoten mit Grad $< k$ existiert.
2. Entfernte Werte kommen nicht in Register, sondern werden in den Speicher ausgelagert. Dazu Lade- und Speicheroperationen erzeugen.
3. Neuer Versuch: Interferenzgraph neu aufbauen und Heuristik erneut anwenden.

Optimistisches Färben



- Problemfall: Offensichtlich 2-färbbar aber kein Knoten mit Grad < 2
- Verbesserung: Optimistisches Färben (Briggs et.al.)
  - Falls kein Knoten mit Grad $< k$ vorhanden, eliminiere Knoten mit Grad k
  - Wenn später keine Farbe frei ist zurück zum Auslagern

Kapitel 9: Registerzuteilung

Constraints: 2-Adress Code

2-Adress Code: Zielregister gleich einem Operandenregister.
Beispiel: **subl** reg0, reg1 hat Semantik reg0 $\leftarrow$reg0 $-$ reg1
Mögliche Modelierungen:

- Zähle Instruktion nicht als Definition sondern nimm Operand. Falls Operand nach Instruktion noch lebendig füge Kopie vor Instruktion ein und zähle diese als Definition.
- Modelliere als 3-Address Code und repariere nach Zuteilung:
  - **op** R0, R1 $\rightarrow$R0 erfüllt 2-Adress Constraints
  - **op** R0, R1 $\rightarrow$R1 falls op kommutativ: **op** R1, R0
  - **sub** R0, R1 $\rightarrow$R1 umformen zu **neg** R1; add R1, R0
  - **op** R0, R1 $\rightarrow$R1 falls op nicht kommutativ $=>$ Problem (Vermeiden durch Einfügen von zusätzlichen Kopien möglich)
  - **op** R0, R1 $\rightarrow$R2 umformen zu **mov** R0 $\rightarrow$R2; **op** R2, R1

Constraints: Aufrufkonventionen, beschränkte Befehle

Teilweise sind nur Teilmengen einer Registerklasse als Operand/Ziel zulässig:

- call-Argumente durch Aufrufkonventionen auf Register festgelegt
- Maschinenspezifische Beschränkungen

Behandlung:

- Nur Teilmengen im Registerallokator erlauben:
  $\Rightarrow$ schränkt eventuell lange lebende Variablen stark ein: unnötiges Auslagern.
- Einfügen von Kopien, kurz vor beschränkter Instruktion. Führt eventuell zu unnötigen Kopien.
- Benutze intelligenten Registerallokator der bei Bedarf Werte zwischen Registern verschiebt.

$\Rightarrow$ Forschungsthema: Beziehung zwischen einfügen von zusätzlichen Kopien („Live Range Splitting“) und Elimination unnötiger Kopien („Copy Elimination“).

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.