Registerallocation

Outline

Einleitung

Was ist Registerallocation und wozu ist sie gut?

- Einer der letzten Schritte der Codegenerierung
- Ausnutzung von Registern für Operationen
- Register schneller als Arbeitsspeicher
- Effiziente Nutzung vorhandener Hardware
- Register nur in begrenztem Maße vorhanden

Theoretische Unterscheidung zweier Schritte

- Registervergabe
- Registerzuweisung

Outline

Lokale Registerallocation

Nur innerhalb von Basisblöcken

- Ununterbrochene Befehlsfolge
- Sprungbefehl nur am Ende eines Blocks
- Sprungbefehle verweisen nur auf Anfang des Blocks

Grenzen via *Boundary Constraints* definiert

- Welche Werte bei Eintritt/Verlassen Live
- Welche Werte in welchem Register
- Information über *gespillte* Werte

Als Teil der globalen Registerallocation durchgeführt

Globale Registerallocation

Arbeitsweise der globalen Registerallocation

- Über Basisblockgrenzen hinaus $\rightarrow$ *global*
- Kennt Struktur (*Flussgraphen*) des Programms
- Erkennen der *Liveness* an Blockgrenzen
- Ausnutzung von Schleifen
- Steuert die lokale Registerallocation
- Definiert *Boundary Constraints* für lokale Registerallocation
- Lokale und global in der Praxis nicht getrennt

Outline

Graph-Coloring-Algorithmus

Ablauf des Graph-Coloring-Algorithmus

- Aufbau des Interferenzgraphen
- Färbung durch Graphenreduktion
- Farbe entspricht Register
- Keine Färbung möglich $\Rightarrow$ *Spill*
  $\Rightarrow$ Erneute Durchführung
- Rekonstruktion des Graphen und *Registerzuweisung*

Interferenzgraph

Kernstruktur des Graph-Coloring-Algorithmus

- Pro *Pseudoregister* ein Knoten
- Operandenpaare $\Rightarrow$ Kante
- Für große Programme ineffizient
  $\Rightarrow$ *duale Repräsentation*
- Bei `N` Knoten eine Symetrische $\texttt{N} \times \texttt{N}$ Bitmatrix
- Nachbarschaftsvektor $\Rightarrow$ *Grad* des Knoten

Interferenzgraph ...

Aufbau in zwei Schritten

Schritt 1: Aufbau der Bitmatrix

- Sequenzielle Auswertung der *Pseudoinstruktionen*
- Für alle Operandenpaare $\Rightarrow$ Eintrag in Bitmatrix

Schritt 2: Nachbarschaftsvektor

- An Knoten die Nachbarschaftsvektoren erzeugen ...
- ... mit Einträgen aus Bitmatrix befüllen
- Länge des Vektors gibt Grad des Knoten an

Färbung des Graphen (ohne *Spilling*)

Färbung durch Reduktion des Graphen:

- Sei `D(K)` Grad des Knoten `K`, `R` Anzahl der Register
- Existiert Knoten `K` mit `D(K) < R`
  $\Rightarrow$ Entferne Knoten aus dem Graphen
- Wiederholen solange Bedingung gilt, oder bis Graph geleert

Färbung des Graphen (mit *Spilling*)

Wenn kein Knoten `K` mit `D(K) < R` existiert:

- Knoten müssen *gespillt* werden
- Für Knoten werden *Spillkosten* ermittelt
- Knoten mit geringsten Kosten wird entfernt
- $\Rightarrow$ wie zuvor fortsetzen
- Entstandenen *Spillcode* schreiben
- Erneute Durchführung $\rightarrow$ Aufbau Interferenzgraph

Ermittlung von Spillkosten

Knoten erzeugen unterschiedlichen Spilloverhead

- Bei Operand $\Rightarrow$ `Load`-Operation
- Bei Resultat $\Rightarrow$ `Store`-Operation
- Häufige Nutzung $\Rightarrow$ hohe Kosten
- In inneren Schleifen erzeugen höchste Kosten
  $\Rightarrow$ als letztes spillen

Die Effizienz stark Abhängig von Auswahl des zu spillenden Werts

Rekonstruktion

Knoten entfernt ohne Erzeugung neuen *Spillcodes*

- Knoten (umgekehrter Reihenfolge) wieder in Graphen eingefügen
- "Freie" Register zuweisen
- Keinem Knoten aus Nachbarschaftsvektor zugewiesen

Outline

Linear-Scan-Algorithmus

Übersicht

- Bietet lineares Laufzeitverhalten
- Basiert nicht auf Graph, sondern auf Liste von Liveintervallen
- Überlappende Intervalle bieten Konfliktsituation
- Bei `R` Registern und `n` gleichzeitigen Intervallen min. `n - R` müssen ausgelagert werden

Liveintervalle

Erzeugung der Liveintervalle

- Basiert auf Liveness der Pseudoregister
- Iteration über Pseudoinstruktionen
- Intervallegrenzen von `R` sind erste und letzte Pseudoinstruktion `[i, j]` in denen `R` live ist
- $\nexists$`i'` mit `i'` $<$ `i` und `R` ist live in `i'` und $\nexists$`j'` mit `j'` $>$ `j` und `R` ist live in `j'`
- Zwischenintervalle möglich, werden jedoch nicht berücksichtig

Registervergabe

Liste `active` mit Liveintervallen, die Registern zugwiesen sind
Chronologisch (nach Startzeitpunkt) alle Liveintervalle bearbeiten

- Abgelaufene Intervalle aus `active` entfernen
- Platz in `active` $\Rightarrow$ aktuelles Inverall aufnehmen
  $\Rightarrow$ aktuellem Interval wird Register zugewiesen
- Kein Platz in `active` $\Rightarrow$ Intervall *spillen*

Intervall *spillen*

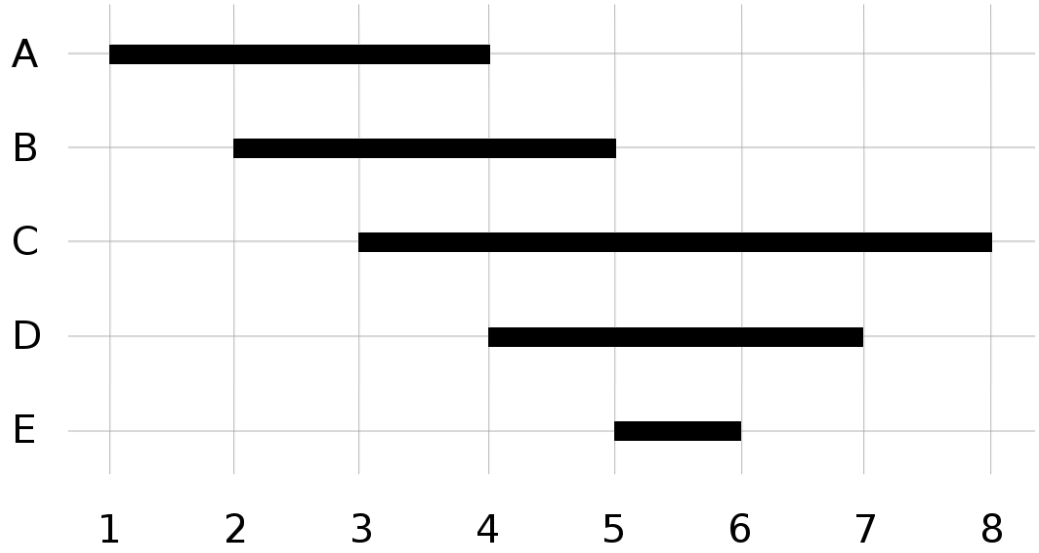
Auswahl des zu spillenden Intervalls

- Intervall zum Auslagen auf Stack wählen
- Wähle Intervall mit spätestem Endpunkt
- Da `active` nach Intervallende sortiert: das Letzte in `active` oder das aktuelle Intervall

Wenn aus `active`

- Aktuellem Intervall wird Register des zu spillenden zugewiesen
- Das zu spillende wird aus `active` entfernt
- Das aktuelle Intervall wird in `active` aufgenommen

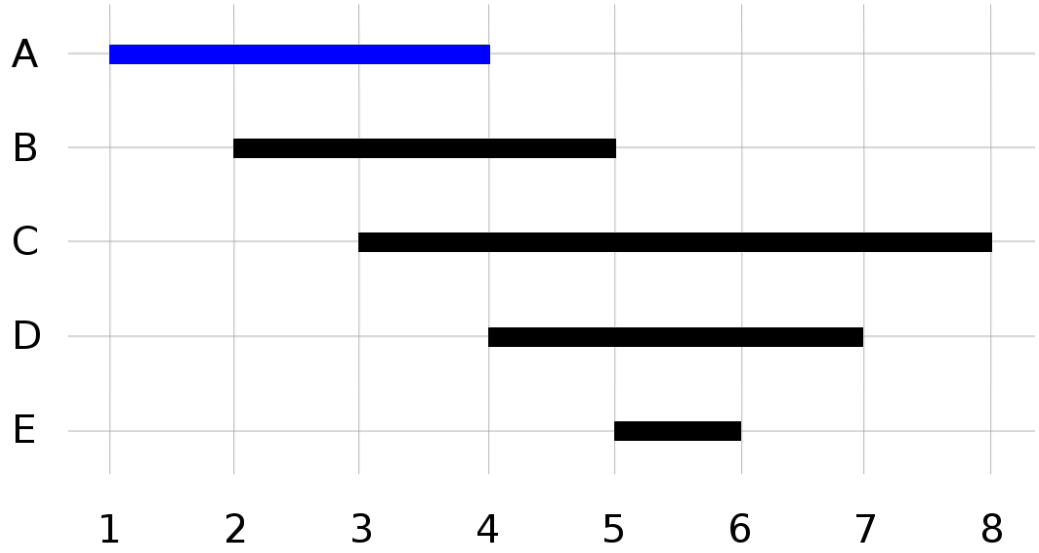
Beispiel




```
active={}
reg0={}
reg1={}
spilled={}
```

```
active={}
reg0={}
reg1={}
spilled={}
```

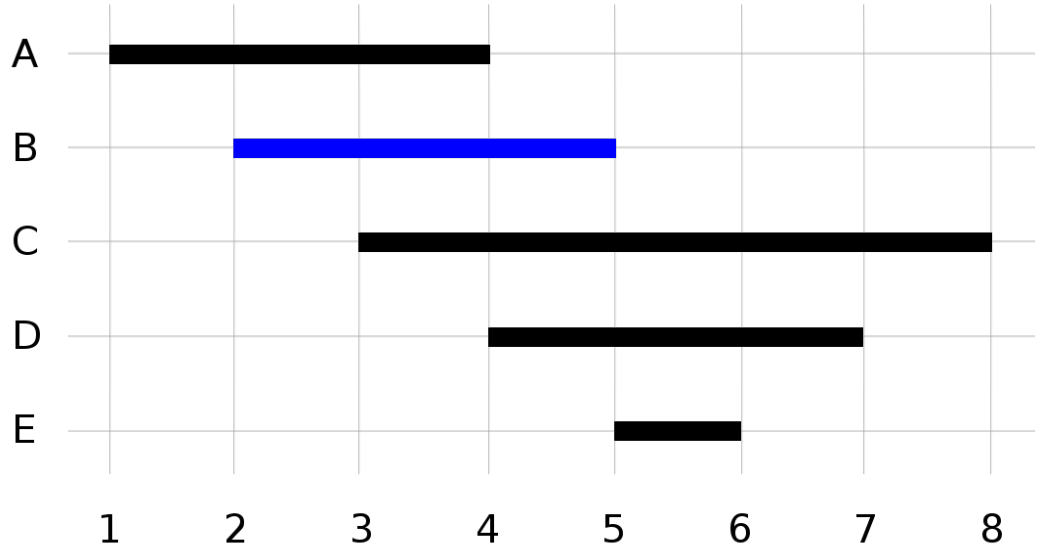
Beispiel




```
active={a}
reg0={a}
reg1={}
spilled={}
```

```
active={a}
reg0={a}
reg1={}
spilled={}
```

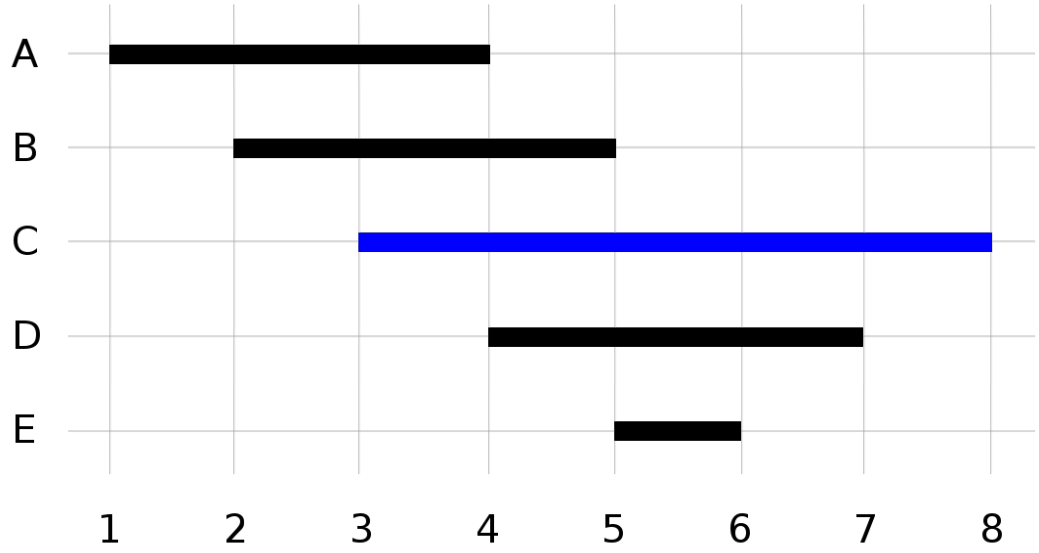
Beispiel




```
active={a, b}
reg0={a}
reg1={b}
spilled={}
```

```
active={a, b}
reg0={a}
reg1={b}
spilled={}
```

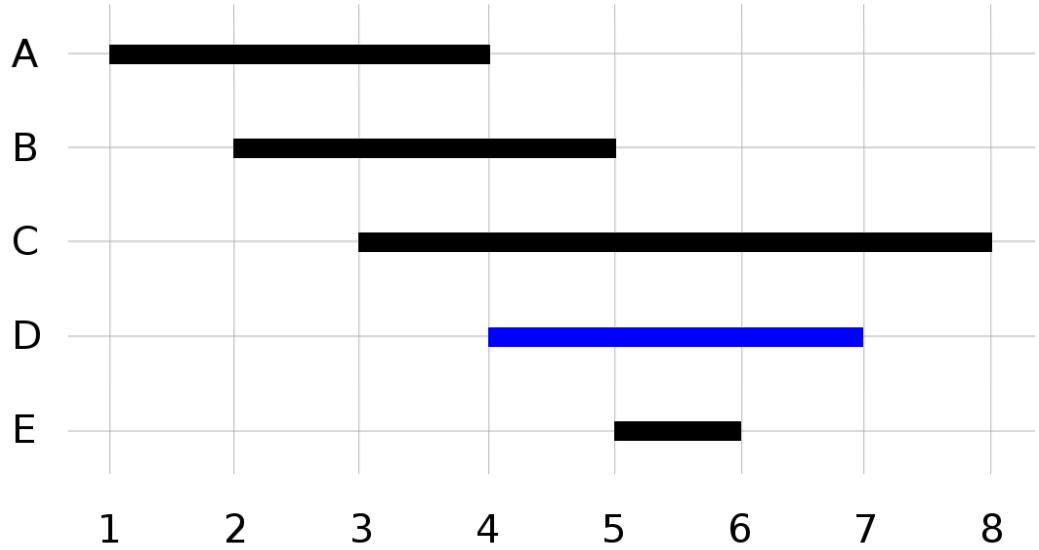
Beispiel




```
active={a, b}
reg0={a}
reg1={b}
spilled={c}
```

```
active={b, c}
reg0={c}
reg1={b}
spilled={a}
```

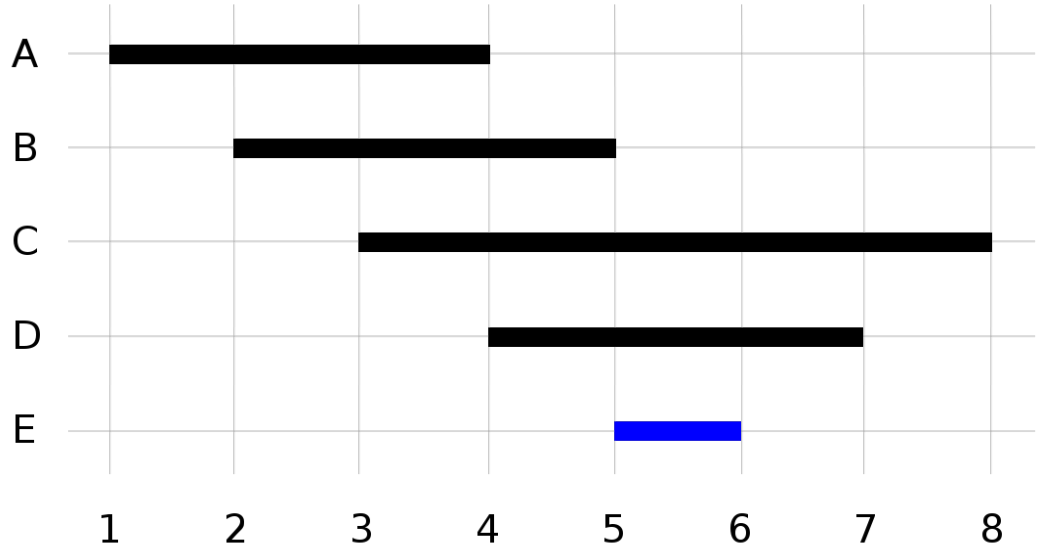
Beispiel




```
active={b, d}
reg0={a, d}
reg1={b}
spilled={c}
```

```
active={d, c}
reg0={c}
reg1={d}
spilled={a, b}
```

Beispiel




```
active={e, d}
reg0={a, d}
reg1={b, e}
spilled={c}
```

```
active={d, c}
reg0={c}
reg1={d}
spilled={a, b, e}
```

Komplexität

Sei `V` Anzahl der Intervalle und `R` Anzahl der Register

- Bei konstantem `R` $\Rightarrow$ $O(V)$
- Bei wachsendem `R` $\Rightarrow$ Einfügen in `active` bedeutender Faktor
- Ist `active` eine einfache Liste $\Rightarrow$ $O(V \times R)$
- Ist `active` ein balancierter Binärbaum $\Rightarrow$ $O(V \times log\ R)$
- Binärbaume erzeugen zusätzlichen Overhead
  $\Rightarrow$ in der Praxis werden Listen genutzt

Outline

Vergleich der Algorithmen

Es müssen zwei Bereiche untersucht werden:

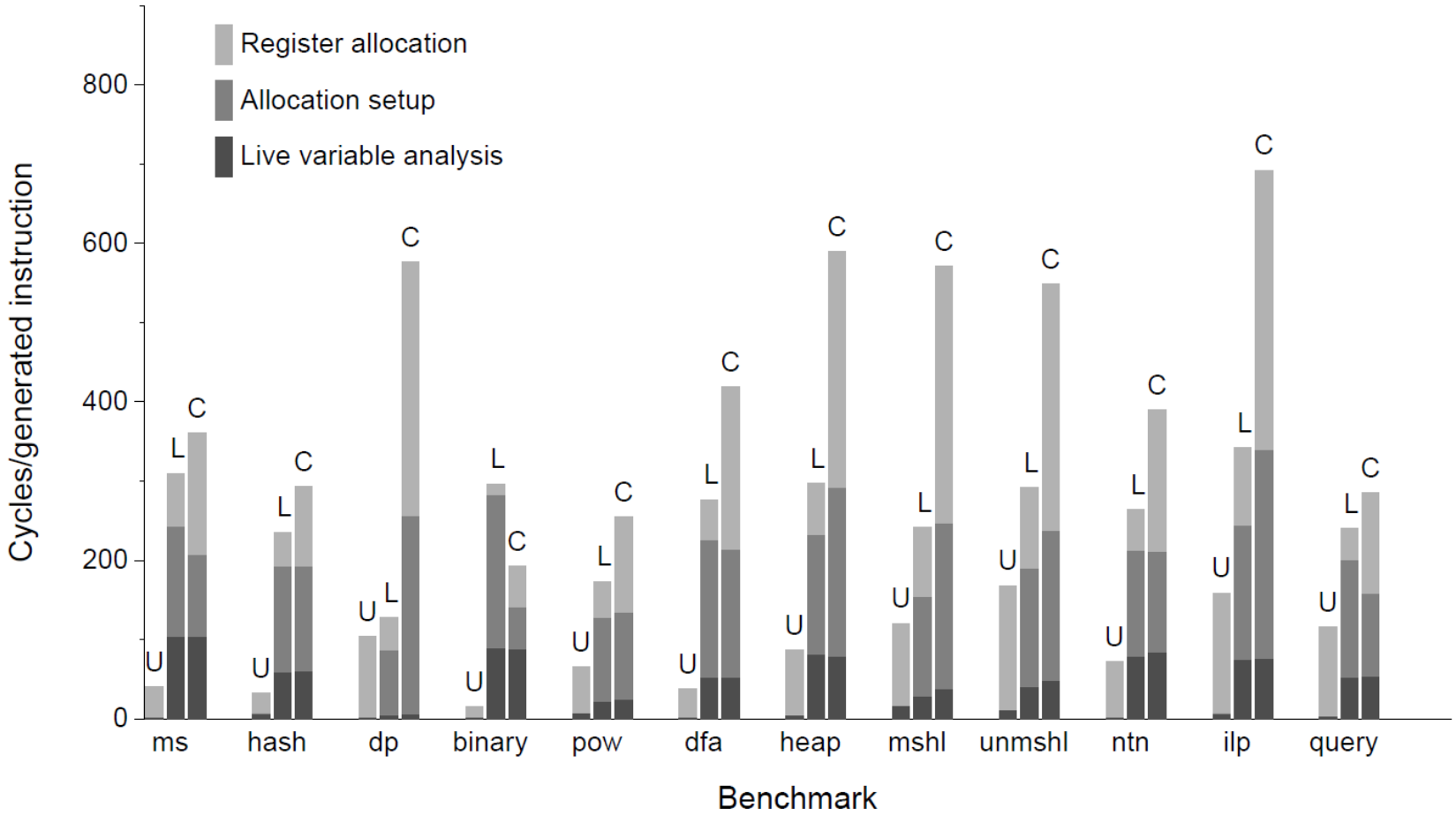
- Laufzeitverhalten des Allocators (Compilezeit)
- Laufzeitverhalten des generierten Codes

Laufzeitverhalten des Allocators

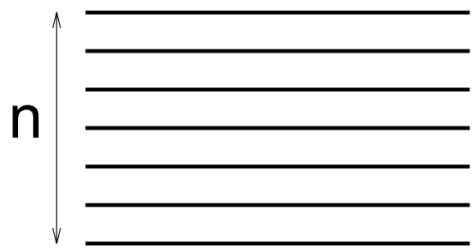
Alle dargestellten Tests wurden mit dem `tcc` Compiler durchgeführt und sind diverse Tests über die folgenden Gebiete:

- Matrixmultiplikationen
- Sortierungsalgorithmen
- diverse andere mathematische Operationen

Laufzeitverhalten des Allocators...



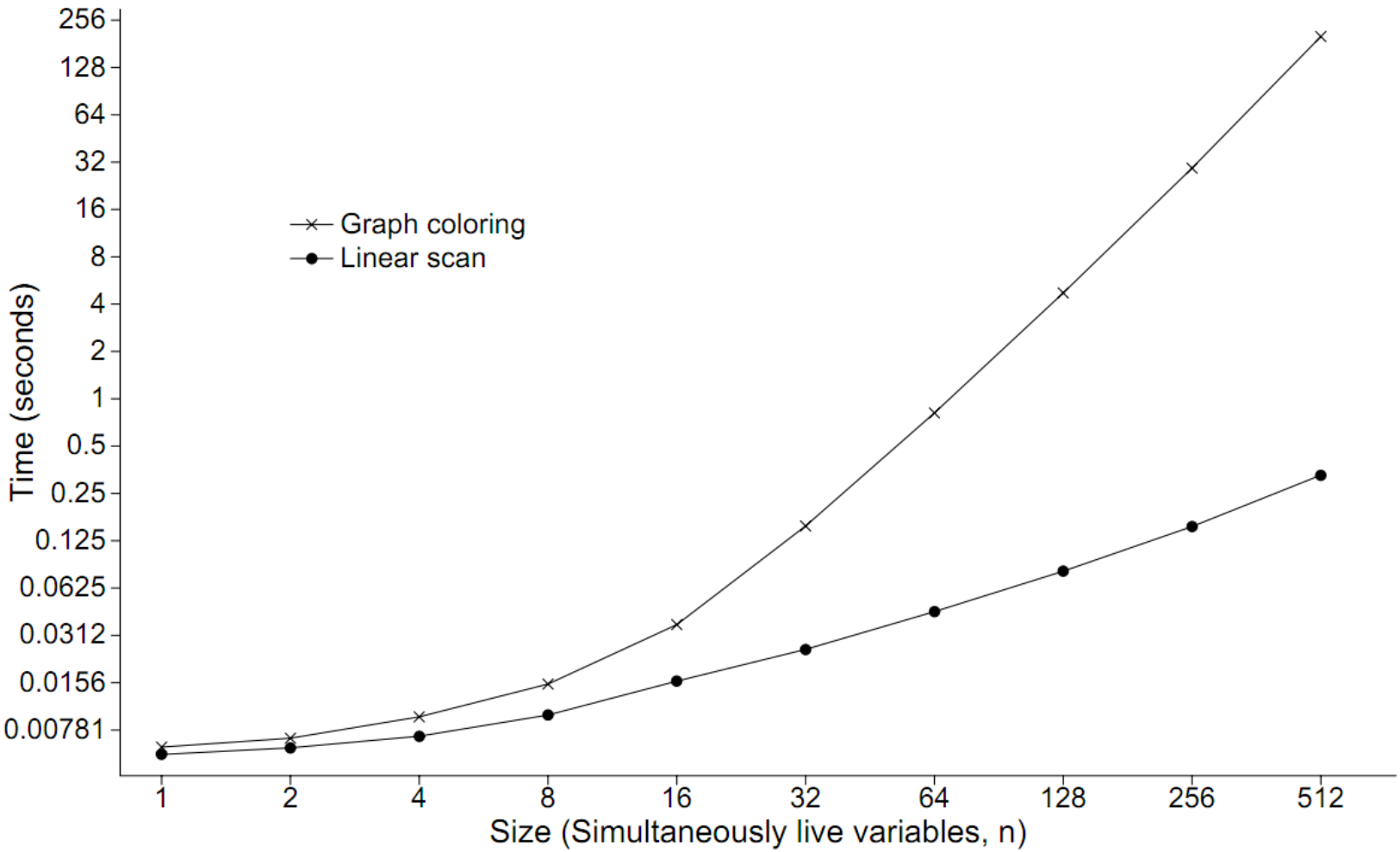
Patologischer Fall



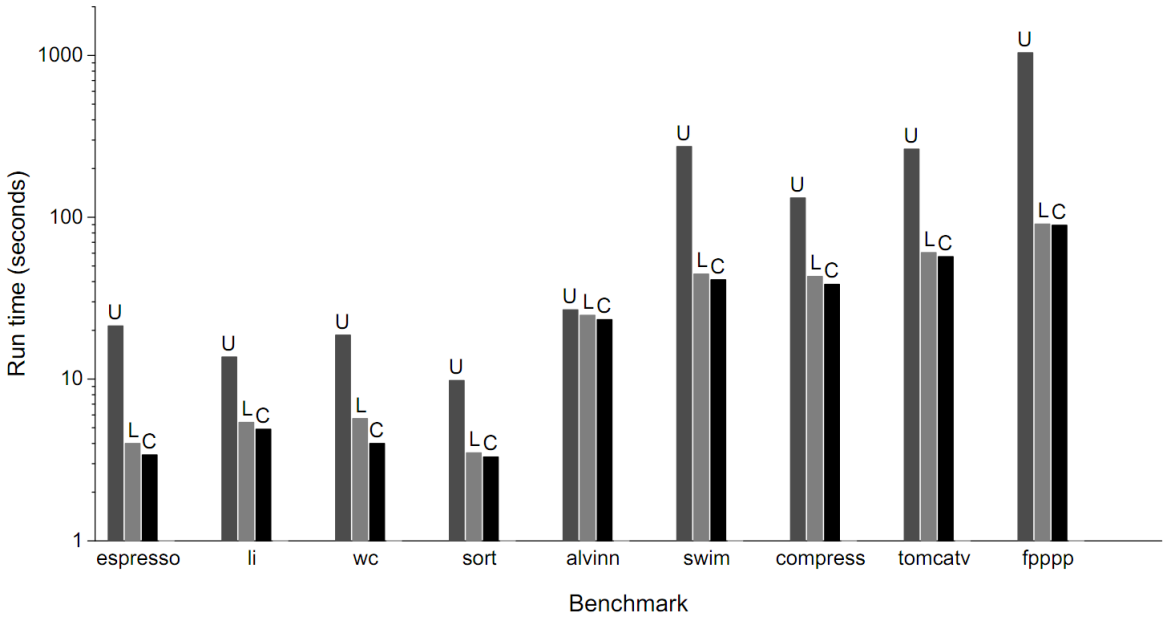
Stresstest des Allocators

- Programm mit `n` gleizeitigen Intervallen
- Bei moderatem `n` $\Rightarrow$ beide Algorithmen annähernd gleich
- Bei stark steigendem `n` $\Rightarrow$ Linear-Scan erheblich schneller als Graph-Coloring

Patologischer Fall...



Laufzeitverhalten des generierten Codes



Programme mit Linear-Scan im Durschnitt 10% langsamer als mit Graph-Coloring

Outline

Fazit

Wann Graph-Coloring-Algorithmus?

- Geschwindigkeit des erzeugten Codes hat oberste Priorität
- Laufzeit des Codegenerators nicht entscheident

Wann Linear-Scan-Algorithmus?

- Geschwindigkeit des Codegenerators hat oberste Priorität
- Geschwindigkeit des erzeugten Codes ist wichtig aber kein K.O.-Kriterium
- Just-In-Time-Compiler