Garbage Collection unter Echtzeitbedingungen

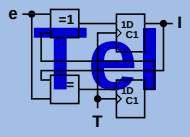
Peter Reichel
peter.reichel@mailbox.tu-dresden.de

Technische Universität Dresden
Institut für Technische Informatik

Inhalt

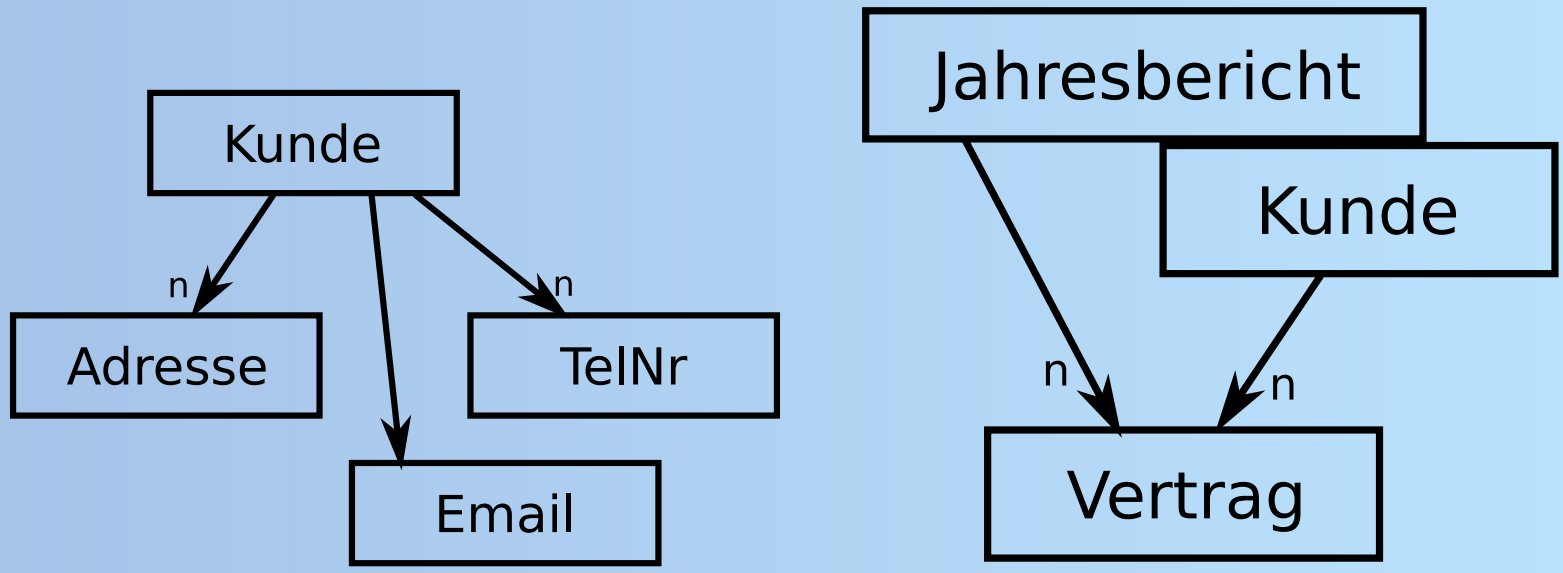
Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

1 Motivation



Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Motivation



- Wann können Objekte gelöscht werden?
- individuelle Lösung aufwändig und fehleranfällig

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

2 Grundlagen der Garbage Collection

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Garbage Collection

Garbage Collection (GC) ist das automatische

- Auffinden und
- Freigeben (wiedernutzbarmachen)

nicht mehr benötigter, allokierter Speicherbereiche.

Im Folgenden:

Objekt	=	allokierter Speicherbereich
Garbage	=	nicht mehr benötigtes Objekt

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Folgen

- Garbage Collection bedeutet eine
  - deutliche Entlastung des Entwicklers
  - verkürzte Entwicklungs- und Testzeiten
  - geringere Fehleranfälligkeit des Systems
- Aber:
  - sehr komplex
  - sehr stark systemabhängig
  - hoher Ressourcenbedarf

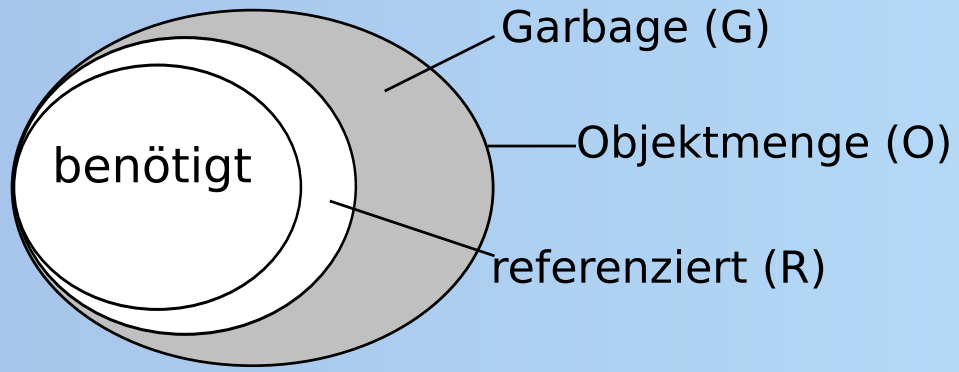
Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

(nicht) benötigt?

- im Wesentlichen ein grundlegender Ansatz:

  Objekt benötigt? $\xrightarrow{Reduktion}$ Objekt referenziert?

- Achtung:
  - nicht referenziert $\longrightarrow$ nicht benötigt $\Rightarrow$ Richtig!
  - nicht benötigt $\longrightarrow$ nicht referenziert $\Rightarrow$ Falsch!
- Entwickler muss Referenzen löschen!

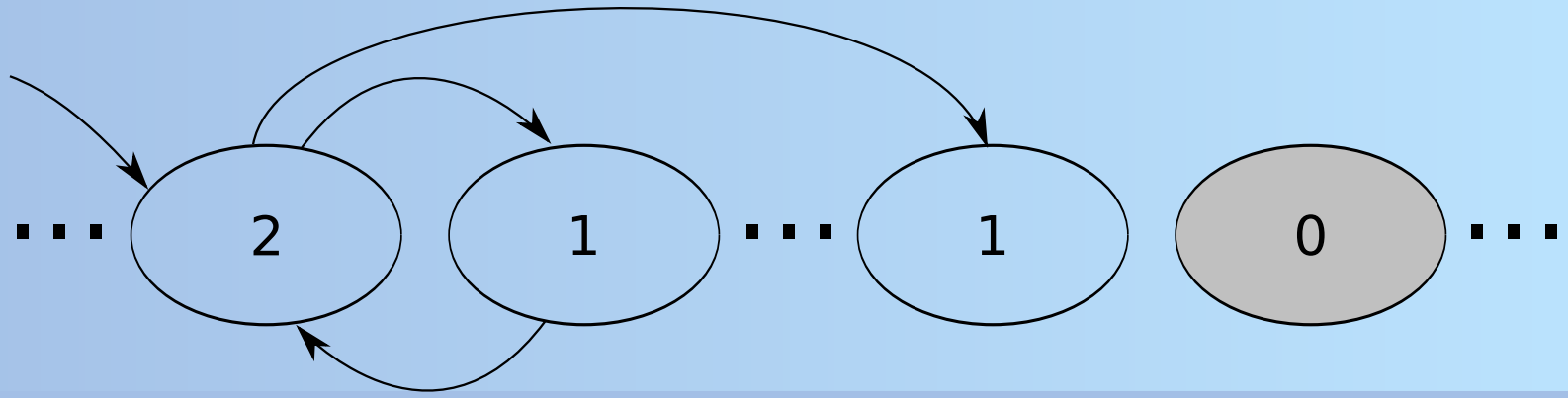


$$G = O - R$$

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Referenziertheit

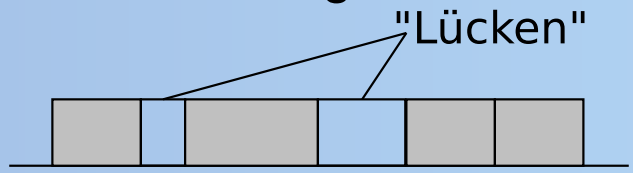
- zwei prinzipielle Herangehensweisen:
  1. Gegenüberstellung einer Invariante (z.B. *reference counting*)
     - Aktualisierung gleichzeitig mit Objektgraphen
     - Rückschlüsse auf Zustand der Objekte
     - zur Laufzeit möglich
  2. Problem der Erreichbarkeit im Graphen
     - Berechnung der transitiven Hülle bzgl. Menge von Root-Objekten die bekanntermaßen erreichbar sind
     - nicht enthaltene Objekte sind Garbage $\Rightarrow$ Freigabe



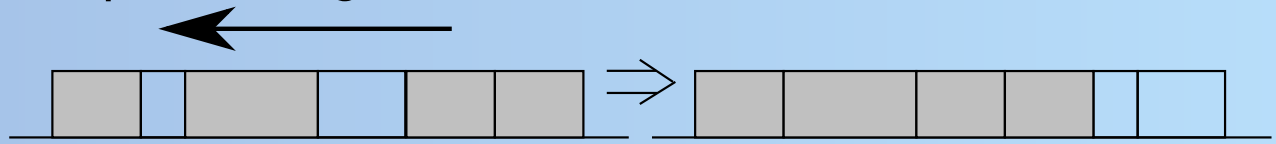
Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Freigabe

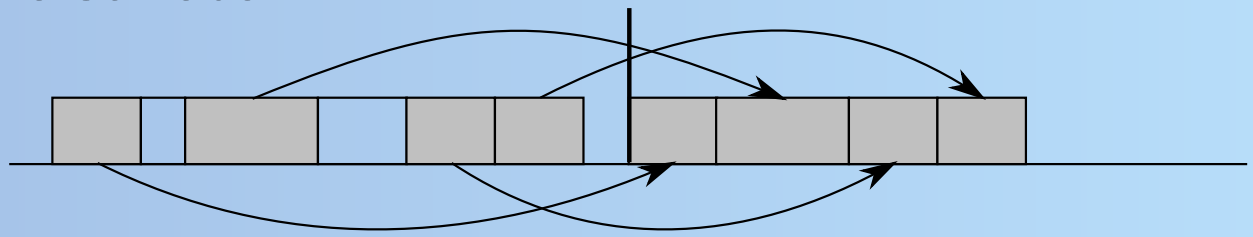
- einfaches Vorgehen



- Kompaktierung



- Verschieben



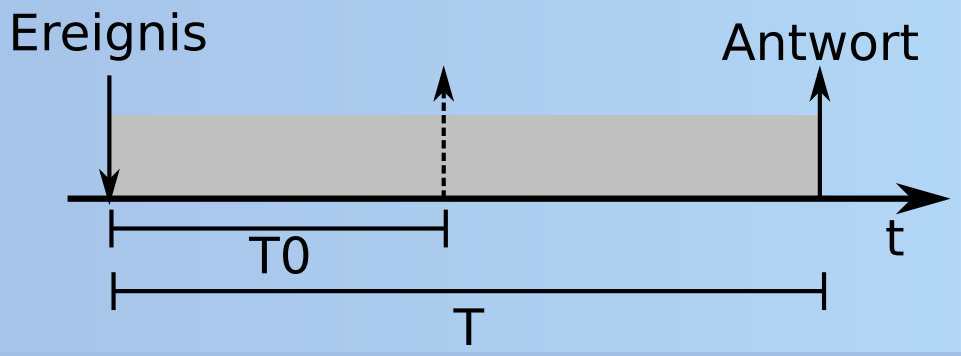
Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

RT (1)

- GC zur Laufzeit der Anwendung (z.B. reference counting):
  - Beispiel:

```
RefAssign(A,B) {
  if (A != null) dec(A);
  if (B != null) inc(B);
  A = B;
}
```

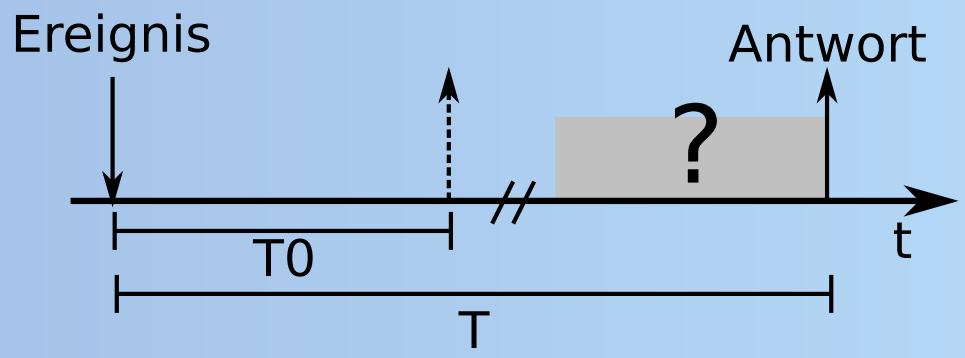
  - zusätzliche Rechenzeit erforderlich
  - zusätzliche Speicherzugriffe erforderlich



Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

RT (2)

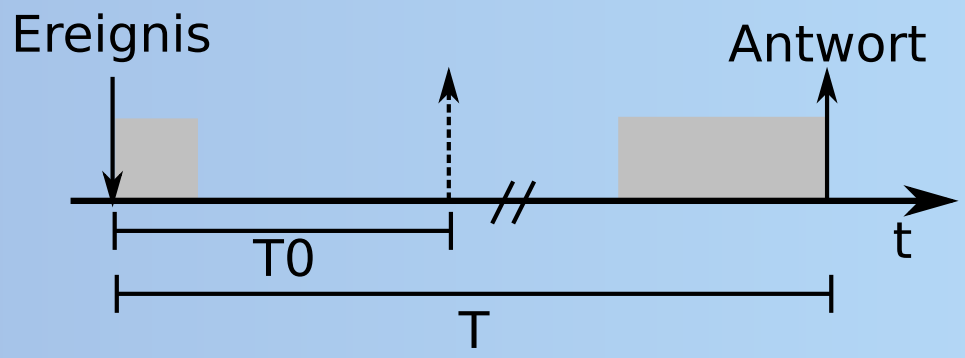
- Unterbrechung der Anwendung für GC:
  - Objektgraph soll sich nicht ändern
  - Unterbrechung für RT jedoch unmöglich!
  - mögliche Lösungen:
    - inkrementelle Garbage Collection
    - nebenläufige Garbage Collection



Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

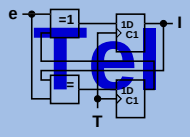
RT Probleme (3)

- begrenzter Speicher / Fragmentierung:
  - Anwendung allokiert ständig Speicher
  - zusammenhängende Blöcke entsprechender Größe erforderlich
    $\Rightarrow$ Allokation in konstanter Zeit
  - GC muss *schnell genug* Speicher freigeben



Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

3 Entwicklung einer MMU für den SHAP-Mikroprozessor



Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Shap

- für MMU und GC wichtige Eigenschaften:
  - mikroprogrammgesteuerte (→ JVM) Stackmaschine
  - Stack und Heap physisch getrennt
  - Objekte lediglich auf Heap
  - kein direkter Speicherzugriff auf Heap, Objekt-Allokation durch MMU
  - Garbage Collection notwendig
  - Echtzeitanforderungen!

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

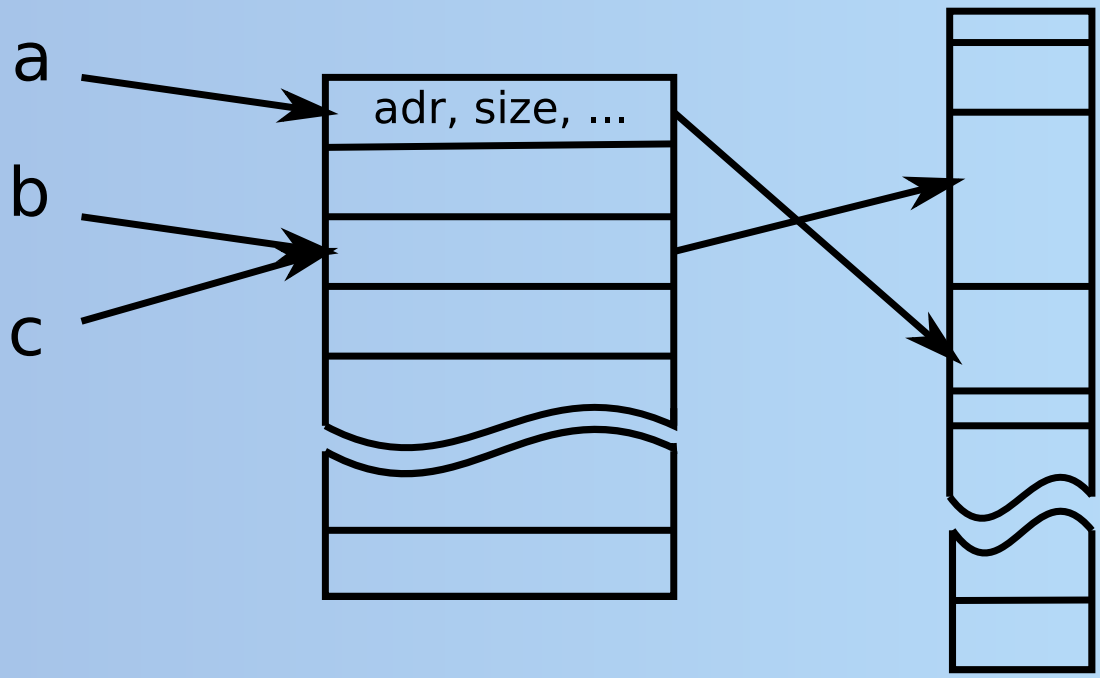
Entwurfsentscheidungen

- Grundlegende Entwurfsentscheidungen:
  - nebenläufige GC auf Basis der Graphenerreichbarkeit ohne Unterbrechung der Anwendung
  - Realisierung des GC als eigenständigen Teil der MMU
  - ermöglichen von Allokation in konstanter Zeit und Verhinderung von Fragmentierung durch Umkopieren in anderen Speicherbereich
  - reine HW-Implementierung

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Speicherverwaltung (1)

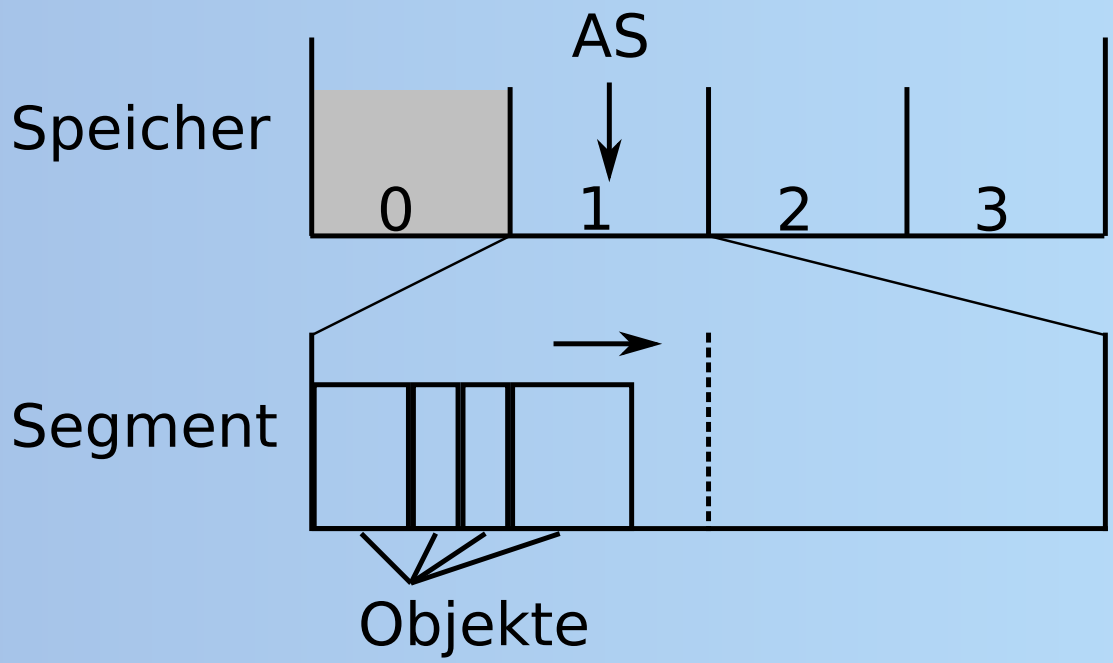
- Verwendung einer zweifachen Indirektion, um Verschieben im physischen Speicher zu ermöglichen
- Zahl der Tabelleneinträge begrenzt
- Begrenzung der Objektgröße



Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Speicherverwaltung (2)

- Aufteilung in 2^n Segmente ermöglicht konstante Allokationszeit
  - Segment mind. doppelt so groß wie max. Objektgröße
  - ein leeres Segment als *Allokations-Segment* (AS) ausgewählt
  - freier Speicher im AS erschöpft $\Rightarrow$ neues AS wählen
  - Zuordnung der Objekte zu den Segmenten



Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Speicherverwaltung (3)

- Segmente ermöglichen effektive Garbage Collection:
  - nach Allokation: Speicherung der Anzahl verbleibender *freier* Wörter
  - bei GC: Speicherung der Anzahl nicht mehr benötigter *toter* Wörter
  - kann freier Speicher gewonnen werden:
    - umkopieren der noch benötigten Objekte eines Segments in andere Segmente
    - Freigabe und Wiedernutzbarmachung des gesamten Segments
    - Freigabe aller nicht benötigten Referenz-Tabelleneinträge

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

GC-Strategie

- Strategie zur Garbage Collection:
  → nebenläufig, unterbrechungsfrei

1. interne Initialisierung
2. Beginn der Stack-Überwachung (exakt durch Markierungs-Bit)
3. Scan des Stacks (exakt)
4. Scan des Heaps ausgehend von durch Stack-Scan und -Überwachung gefundenen Root-Objekten (momentan konservativ)
5. Ende der Stack-Überwachung
6. Markieren aller nicht erreichbaren Objekte als Garbage und Aktualisierung der Segment-Statistik
7. Freigabe auf Segmentebene

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Momentaner Stand

- funktionsfähig:
  - Verwaltung von Referenzen und Segmenten
  - Stack-Überwachung, Stack-Scan und Heap-Scan
  - Markierung nicht erreichbarer Objekte als Garbage und Aktualisierung der Segment-Statistik
- noch offen:
  - geeignete Auswahlfunktion zur Bestimmung ob umkopieren sinnvoll
  - umkopieren und freigeben der Segmente
  - eingehende Effizienzbetrachtungen

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Ausblick

- Prüfung ob konservative Strategie beim Heap-Scan durch exakten Ansatz ausgetauscht werden kann
- Verkürzung der Gesamtzeit der GC durch Nutzung generationeller Prinzipien auf Basis des Konzepts der Segmente
- Berücksichtigung von Shared-Memory im Hinblick auf weitere Entwicklung des SHAP-Prozessors

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de

Danke

Vielen Dank für Ihre Aufmerksamkeit!

Peter Reichel
Technische Universität Dresden
Institut für Technische Informatik
peter.reichel@mailbox.tu-dresden.de