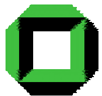
Plattformunabhängiges Programmieren von Rechnerbündeln

Seminar Dagstuhl im Oktober 2002



UNIVERSITÄT KARLSRUHE (TH)
Fakultät für Informatik

Bernhard Haumacher
Institut für Programmstrukturen
und Datenverarbeitung · Prof. Tichy

Ziele

- Modulare Erweiterung einer parallelen objektorientierten Sprache für die verteilte Umgebung
  - Programmiermodell
    - Java mit mehreren Kontrollfäden
    - Berücksichtigung der Verteilung: Erweiterung des Objektmodells
  - Effizienz
    - So schnell wie der beste verfügbare JIT-Übersetzer
    - Verschiedene Kommunikationstechnologien

Lösungsansatz

- Programmtransformation
  - Ausgangspunkt: Erweitertes Java
  - Endprodukt: Standard-Java + Bibliotheksaufrufe
  - Modularität: standardkonforme virtuelle Maschine verwendbar
  - Effizienz: VM mit bestem JIT benutzbar.
- Bibliotheksunterstützung
  - Effizienz: Angepasste Bibliotheken, wo der Standard nicht ausreicht (Netzwerkanbindung!)

Überblick Gesamtsystem – 1

- “Verteilte virtuelle Java Maschine”
  - Objektmodell
    - lokale Objekte
      - auf Erzeugungs-VM beschränkt
      - Wertsemanik bei *globalen* Operationen
    - globale Objekte
      - Objekte der verteilten virtuellen Maschine
      - Referenzsemantik
      - Lokalität
        - Entfernt: Auf dediziertem Knoten, Fernaufruf
        - Repliziert: Überall lokal, Konsistenzoperationen

Überblick Gesamtsystem – 2

- "Verteilte virtuelle Java Maschine"
  - Parallelitätsmodell
    - Maschinenüberspannende Kontrollfäden
      - Erhaltung der Thread-Semantik von Java für die verteilte virtuelle Maschine
  - Basistechnologie
    - Effizienter Fernaufruf
    - Steckbare Kommunikationstechnologie
      - Myrinet/ParaStation, Myrinet/GM, TCP/IP
    - Schnelle Objektserialisierung

Transparent Replizierte Objekte

UNIVERSITÄT KARLSRUHE (TH)
Fakultät für Informatik

Bernhard Haumacher
Institut für Programmstrukturen
und Datenverarbeitung · Prof. Tichy

Annahmen

- Verhältnis Schreib-/Leseoperationen << 1 bei replizierten Objekten
  - ansonsten: entferntes Objekt verwenden
- Solange Referenz auf Replikat existiert, werden Werte auf dem Knoten benutzt.
  - ansonsten: Aufräumen durch den DGC

Folgerungen / Anforderungen

- Leseoperationen ungebremst schnell lokal
- Seltene Schreiboperationen tragen Mehraufwand für Replikation.
- Strategie für Konsistenzhaltung:
  - Anders als viele “native” DSM-Systeme
    - Invalidieren und Aktualisieren bei Bedarf als Ersatz für fehlenden Speicherbereiniger
  - Hier: Bring-Strategie besser geeignet.

Anforderungen: Transparenz

- Erzeugung analog zu entfernten Objekten
  - new Constructor(arg1, ...)
- Implizite Erzeugung und Freigabe von Replikaten
  - Erzeugung: Übergabe eines replizierten Objektes in einem entfernten Methodenaufruf
  - Freigabe: durch verteilten Speicherbereiniger
- Zugriff stets auf das lokale Replikat
- Konsistenzoperationen bei Synchronisation

Entwurf: Referenzen

- Referenz ist gleichzeitig Replikat
  - keine Indirektion, schneller lokaler Zugriff.
- Nur eine Referenz pro VM
  - anders als bei entfernten Objekten
  - vermeidet unnötigen Aufwand bei Konsistenzoperationen
  - Mehraufwand bei Referenzübergabe durch Auffinden eines existierenden Replikats

Entwurf: Speicherbereiniger

- Erweiterung des verteilten Speicherbereinigers
  - Problem:
    - Original benötigt Referenz auf seine Replikate
    - Aber: Von der Applikation nicht mehr referenzierte Replikate müssen abgeräumt werden.
  - Lösung:
    - Schwache entfernte Referenzen
      - vom Speicherbereiniger verwaltet
      - verhindern das Abräumen nicht

Entwurf: Konsistenz – 1

- Standard-Java (Java Memory Model)
  - Freigabe Konsistenz
    - Thread-lokaler Cache für den Heap
      - Leeren beim Betreten einer Synchronisation
      - Zurückschreiben vor dem Verlassen einer Synchronisation
  - Seqenzielle Konsistenz
    - für volatile deklarierte Variablen
      - Änderungen werden für andere Threads in der Reihenfolge ihrer Verursachung sichtbar.

Entwurf: Konsistenz – 2

- Versuch: direkte Übertragung
  - Replikat: lokaler Cache für entferntes Original
  - Invalidieren beim Betreten einer Synchronisation
  - Bei Benutzung: Füllen des Replikats mit Werten
  - Zurückschreiben von Änderungen vor Verlassen einer Synchronisation.
  - Zugriffe auf volatile deklarierte Variablen direkt entfernt in global eindeutiger Reihenfolge
- Ermöglicht keine effiziente Umsetzung

Entwurf: Konsistenz – 3

- Lesesynchronisation lokal am Replikat
  - Änderungen am Replikat werden Java-konform mit Eintritt in die Synchronisation sichtbar
  - Lesen auf unterschiedlichen Knoten nebenläufig möglich
- Schreibsynchronisation sperrt alle Replikate
  - Aufzeichnung von Änderungen während des Synchronisierten Abschnitts
  - Nachziehen der Änderungen auf den Replikaten vor der Freigabe

Problem: Unterscheidung Schreiben / Lesen

- Leseoperation

```
synchronized (r) {
    // Leseoperationen
    // auf r
}
```

- Schreiboperation

```
synchronized (r) {
    // Schreiboperationen
    // auf r
}
```

Java kennt keine Unterscheidung zwischen Schreib- und Lesesperren.

Lösungsansätze: Schreib- / Leseunterscheidung

- Automatisch
  - Übersetzer sucht nur-lese Eigenschaft der synchronisierten Operationen nachzuweisen
- Optimistisch
  - Annahme: nur Leseoperationen, Rücksetzen bei der ersten Schreiboperation
- Manuell
  - Einführung einer Schreib- und Lesesynchronisation in die Sprache

Problem: Nebenläufige Änderungen – 1

- Standard Java: Keine Übereinstimmung zwischen Synchronisationsobjekt und Objekt dessen Zustand betroffen ist.

```
synchronized (p) {        ||  synchronized (q) {
    // access r           ||      // access r
    r.a = ...             ||      r.b = ...
}                         ||  }
```

- Nebenläufige Änderungen von unterschiedlichen Teilen eines Objektes

Problem: Nebenläufige Änderungen – 2

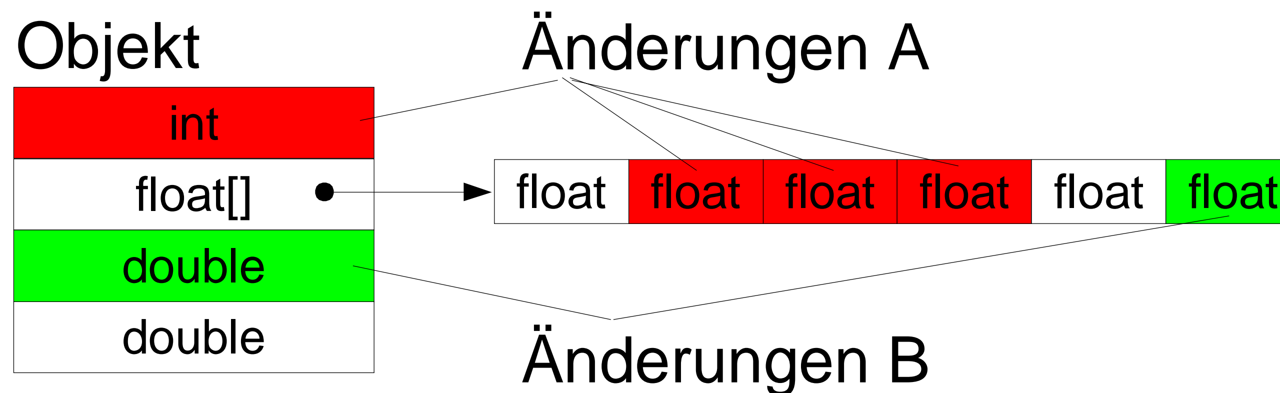
- Änderung an repliziertem Objekt benötigt Sperre an repliziertem Objekt
  - Sperre für Änderung und Sperre für das Beobachten der Änderung muss sich wechselseitig ausschließen
    - Ansonsten: Beobachtung von Teiländerungen.
  - Lesesperre muss lokal möglich sein, sonst macht Replikation keinen Sinn.
  - Zugehörige Schreibsperre benötigt alle möglichen Lesesperren!

Lösung: Nebenläufige Änderungen – 1

- Besitz einer Schreibsperre an einem replizierten Objekt berechtigt zum Ändern von mehreren replizierten Objekten.
- Freigabe einer Schreibsperre löst Konsistenzoperationen auf allen modifizierten replizierten Objekten aus.
- Synchronisation an lokalen und entfernten Objekten bleibt unverändert.
  - Keine Einfluss auf Geschwindigkeit

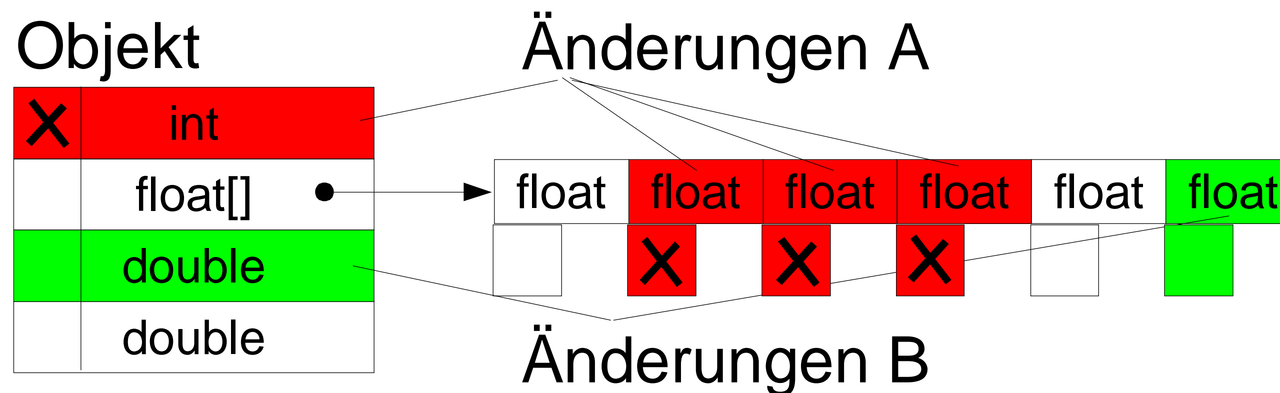
Problem: Aufzeichnung / Detektion von Änderungen

- Minimierung der zu kommunizierenden Daten beim Verlassen der Schreibsperre.
- Kein Überschreiben von nebenläufigen Änderungen beim Nachziehen



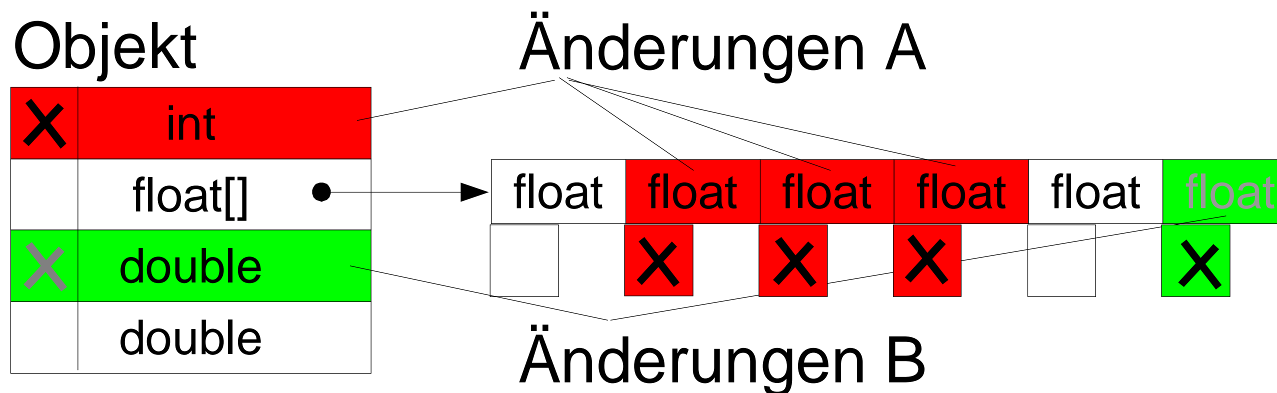
Lösung: Aufzeichnung von Änderungen (partiell) – 1

- Transformation erzeugt pro Eintrag eine zusätzliche Markierung.
- Markierung wird bei Modifikation gesetzt.
- Nur markierte Felder werden übertragen.



Lösung: Aufzeichnung von Änderungen (partiell) – 2

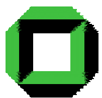
- Markierungen müssen Thread-lokal sein.
  - Nebenläufige Änderungen in der selben VM könnten teilweise frühzeitig sichbar werden.
  - Der falsche Thread würde die Änderungen inkonsistent zu den Replikaten propagieren.



Maschinenüberspannende Kontrollfäden

Remote Monitor Access (RMA)

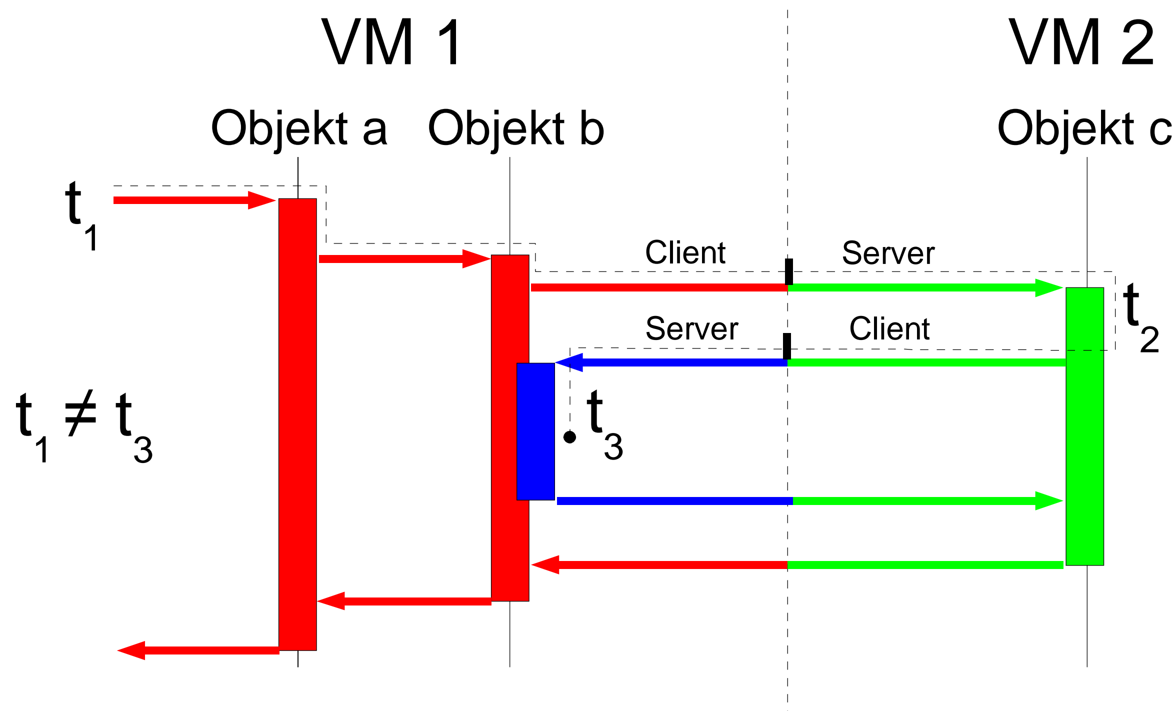
Bernhard Haumacher
Institut für Programmstrukturen
und Datenverarbeitung · Prof. Tichy



UNIVERSITÄT KARLSRUHE (TH)
Fakultät für Informatik

Maschinenüberspannende Kontrollfäden mit RMI

- Abbildung einzelner Segmente auf neue Java-Threads.

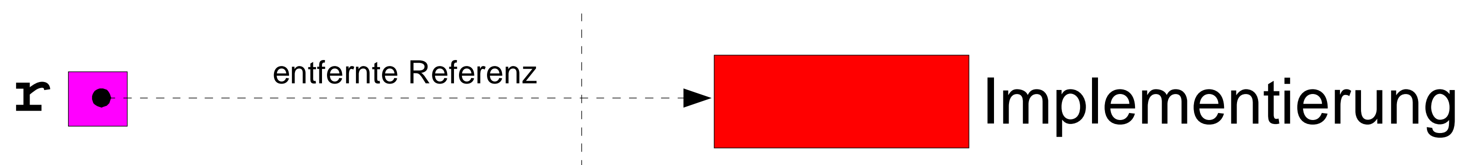


Problem: Synchronisation – 1

- Unerwartetes Verhalten, falls **`r`** entfernte Referenz

```
synchronized (r) {
    // Block
}
```

  - Synchronisation am Stellvertreterobjekt
    - nebenläufiger Zugriff
    - fehlschlagende Thread-Benachrichtigung



Problem: Synchronisation – 2

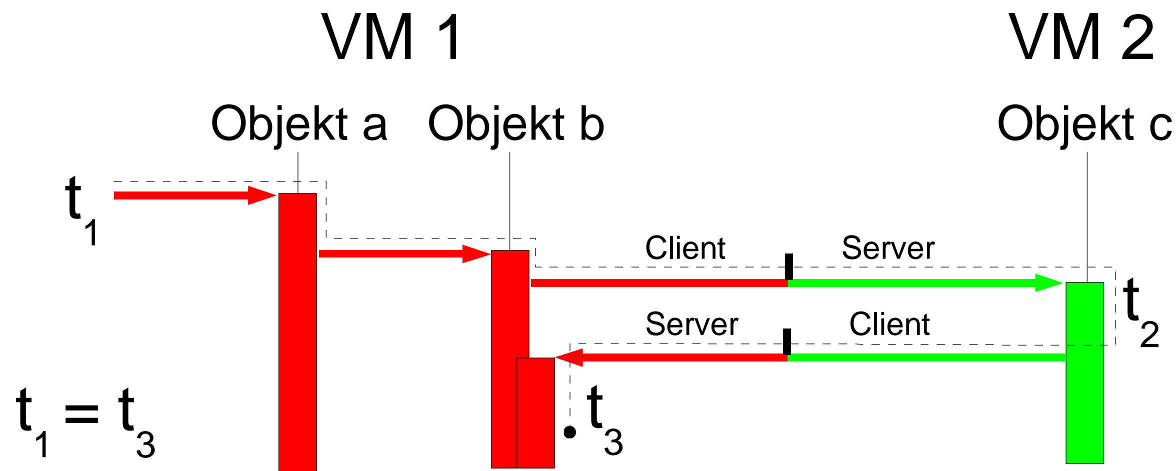
- Anwendung: Schreibsperre an repliziertem Objekt benötigt alle Sperren der Replikate.
- Umschreiben nicht möglich in

```
r.beginSynchronized();    // monitorenter
   // Block
r.endSynchronized();      // monitorexit
```

- Monitoranforderung und -freigabe an syntaktischen Block gebunden
  - beginSynchronized() und endSynchronized() mit RMI nicht implementierbar.

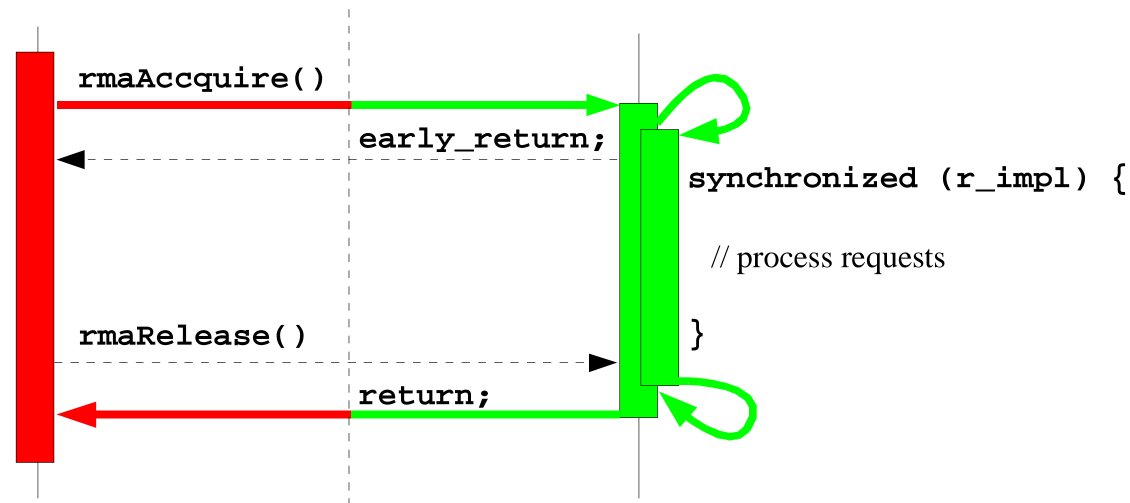
Lösung: RMA – 1

- Eindeutiger Stellvertreter-Thread pro VM für maschinenübespannenden Kontrollfaden
  - Abarbeitung aller Segmente.
  - Zugriff auf entfernten Monitor.



Lösung: RMA – 2

- Trick: Aufruferseitig steuern 2 Methodenaufrufe eine einzige dienstgeberseitige Methode mit frühzeitiger Rückgabe



Lösung: Transformation – 3

- Synchronisierter Block lässt sich umschreiben in RMA-Operationen.

```
synchronized(r){
    // Block
}
```

```
rma = rmaAquire(r);
try {
    // Block
}
finally {
    rmaRelease(rma);
}
```

Lösung: Transformation – 4

- Eindeutige Repräsentanten und RMA liefern
  - Reentrante entfernt synchronisierte Blöcke
  - Transparente machinenüberspannende Kontrollfäden bezüglich Synchronisation.
- RMA: notwendige Voraussetzung für Schreibsperren auf replizierten Objekten.
  - Entfernte Anforderung und Freigabe aller Replikat-Monitore

Ende

Sicherheitskopie und Abgleich

- Kopie vor der Schreiboperation
  - a) beim Eintritt in die Synchronisation
  - b) beim ersten Schreibzugriff
- Alle Operationen auf der Kopie
  - Auch Leseoperationen müssen aktuellen Zustand sehen.
- Berechnen und Verschicken der Differenz vor dem Verlassen der Synchronsisation

Probleme bei nebenläufigen Änderungen

- Bei unverändertem Original ist die Differenz exakt.
- Problem wenn
  - Original von nebenläufiger Änderung modifiziert
  - Änderung bereits sichtbar
- Die berechnete Differenz macht die nebenläufige Änderung rückgängig.