

Universität Karlsruhe (TH)

Forschungsuniversität · gegründet 1825

# Software Transactional Memory

Prof. Dr. Walter F. Tichy

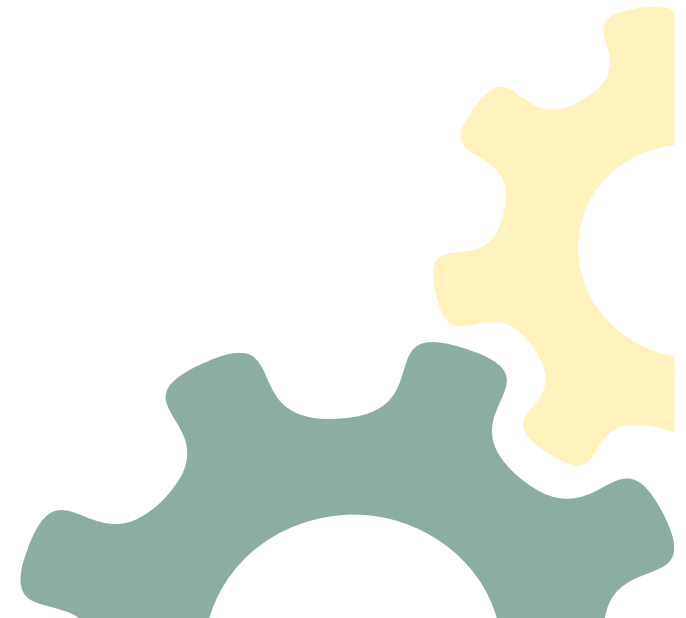
Dr. Victor Pankratius

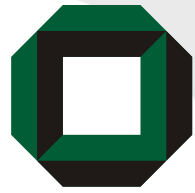
Ali Jannesari



Fakultät für **Informatik**

Lehrstuhl für Programmiersysteme

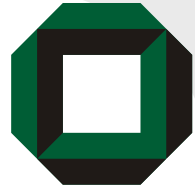




# Software Transactional Memory

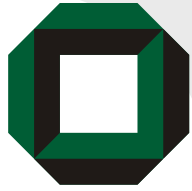
## Agenda

1. Motivation
2. Überblick
3. Konzepte
  - Konfliktdetektion
4. Implementierungen
  - Beispiel einer STM-Implementierung in Java
5. Probleme mit STM
6. Ausblick



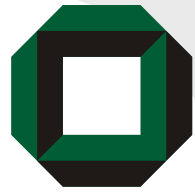
## Motivation (1)

- Der **traditionelle Ansatz zur Synchronisation** mehrfädiger Programme ist die Verwendung von **Sperren**.
- **Probleme**
  - Fehleranfälligkeit beim Programmieren (z.B. Inkonsistentes Lesen/Schreiben, Deadlocks, etc.)
  - Zu viele Sperren (z.B. bei Anfängern, die „auf Nummer sicher“ gehen wollen) führen zu niedriger Performanz aufgrund von wartenden Prozessen



## Motivation (2)

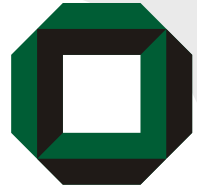
- **Problem-Potenziale beim Prozessabbruch**
  - Werden die Sperren freigegeben?
  - In welchem Zustand befinden sich die gemeinsam genutzten Speicherbereiche?
- **Prioritätsumkehr**
  - Prozess A mit hoher Priorität kann nicht ausgeführt werden, weil ein Prozess B niedriger Priorität eine benötigte Ressource blockiert. Wird B durch einen anderen Prozess C mittlerer Priorität verdrängt, so wird C vor A ausgeführt.
  - Prozesse hoher Priorität sollten nicht auf solche mit niedriger Priorität warten müssen (wichtig insb. bei Echtzeit-Systemen).
- **Sperren-Konvoi („Lock convoying“)**
  - Prozesse arbeiten quasi im Gleichschritt, stürzen sich alle gleichzeitig auf die nächste Sperre.



# Software Transactional Memory (1)

## Überblick

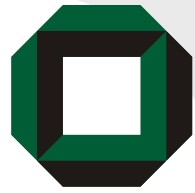
- Beim „Software Transactional Memory“ (STM) wird das **Transaktionskonzept** auf gemeinsam genutzten Speicher übertragen.
- **Transaktion:** Folge von Aktionen mit folgenden Eigenschaften:
  - **Atomarität (Atomicity):** Aktionen werden entweder ganz oder gar nicht ausgeführt
    - bei Erfolg → Abschluss mit „**commit**“ (Änderungen werden wirksam)
    - bei Fehlschlag → Abschluss mit „**abort**“ (Änderungen werden verworfen)
  - **Konsistenz (Consistency):** Zustände vor und nach Ausführung einer Transaktion konsistent. (Beispiele applikationsspezifisch: gemeinsamer Speicher, Invarianten, etc.)
  - **Isolation:** Eine Transaktion „sieht“ andere Transaktionen nicht; sie produziert korrektes Ergebnis, unabhängig von den anderen Transaktionen



# Software Transactional Memory (2)

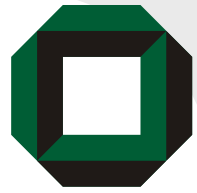
## Überblick

- Transaktionaler Speicher als Alternative
  - Statt Sperren kommen Transaktionen als Koordinationsmechanismus zum Einsatz
  - Transaktionen nehmen dem Programmierer die Entscheidung ab zwischen der Verwendung von
    - wenigen, grob-granularen Sperren
      - Programme sind leichter verständlich, gut nachzuvollziehen
      - Performanz ist nicht optimal
    - vielen fein-granularen Sperren
      - nicht gut überschaubar, fehleranfällig
      - bessere Performanz



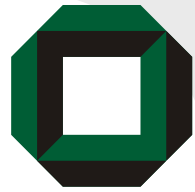
# Transaktionaler Speicher: Konzepte (1)

- Paralleler Zugriff auf Datenstrukturen kann nicht mehr „blockieren“
  - Keine Sperren mehr, d.h. kein Faden kann eine Sperre halten, die globalen Fortschritt verhindert
- Eigenschaften von Synchronisationsmechanismen [He+03, Her91]
  - „**Lock-free**“: Irgend ein Faden macht immer Fortschritte
  - „**Wait-free**“: Jeder Faden macht Fortschritte, sogar wenn andere Fäden ihre Ausführung verzögern oder abbrechen.
    - „stärkere“ Eigenschaft als „lock-free“
- Realisierung von Transaktionen basiert typischerweise auf atomaren Synchronisationsprimitiven, die von der Hardware angeboten werden
  - z.B. **Compare-and-Swap**  
(kann für Implementierungen mit wait-free-Eigenschaft genutzt werden, vgl. [He+03])



# Transaktionaler Speicher: Konzepte (2)

- **CAS: Compare-and-swap**
  - Atomare Operation, die von der Hardware angeboten werden muss.
  - Grundlage für Sperrmechanismen
  - Drei **Argumente**:
    - Adresse einer Hauptspeicherstelle
    - Vergleichswert  $v$
    - neuer Wert  $w$
  - **Funktionsweise** (atomar!)
    - Der Wert an der adressierten Speicherstelle wird mit  $V$  verglichen.
    - Nur bei Übereinstimmung wird  $w$  an dieselbe Speicherstelle geschrieben.
    - Erfolg oder Nichterfolg wird zurückgemeldet, z.B. durch Rückgabe des vorgefundenen Werts.



# Transaktionaler Speicher: Konzepte (3)



*Beispiel: Auszahlung am Geldautomaten  
Der Saldo des Kontos darf nicht **negativ** werden.*

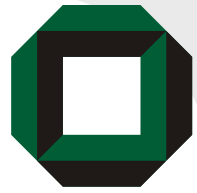
## • **Traditionelles Vorgehen**

- Während des Auszahlvorganges wird eine **Sperre** auf das Konto angefordert. Andere Geldautomaten müssen **warten**, wenn sie gleichzeitig auf dasselbe Konto zugreifen wollen.

## • **Variante ohne Sperren:**

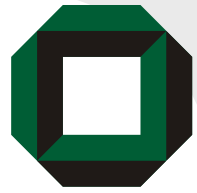
- (1) lese den alten Wert, prüfe die Bonität (und breche evtl. ab)
- (2) ziehe Auszahlungsbetrag ab
- (3) setze neuen Betrag mittels **CAS(Konto, alt, neu)**; Falls das fehlschlägt, starte wieder bei (1).

- Dieser Algorithmus ist „**lock-free**“ aber nicht „**wait-free**“, denn durch andere Zugriffe auf dieses Konto könnte sich der Saldo stets wieder geändert haben, die CAS-Operation fehlschlagen und so den Auszahlungsvorgang **hinauszögern**.



# Transaktionaler Speicher: Konzepte (4)

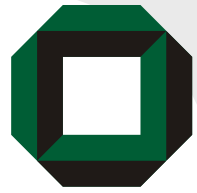
- **Optimistisches Grundprinzip:** Konkurrierender Zugriff auf gemeinsam genutzte Speicherbereiche wird nicht im **Voraus** verhindert, sondern **nachträglich** detektiert.
- Jeder Prozess führt während einer Transaktion Schreib- oder Leseoperationen auf dem gemeinsam genutzten Speicher **zunächst ohne Rücksicht** auf andere Prozesse durch.
- Sobald festgestellt wird, dass ein **Konflikt** aufgetreten ist, wird eine der betreffenden Transaktionen abgebrochen („abort“) und **neu gestartet**.
  - **Konflikt:** Die Daten, auf die lesend oder schreibend zugegriffen wurde und wird, wurden währenddessen durch eine andere Transaktion verändert.
- Wenn **kein** Konflikt aufgetreten ist, wird die Transaktion **abgeschlossen** und mögliche Änderungen werden permanent gemacht („commit“).



# Transaktionaler Speicher: Konzepte (5)

- **Abbruch einer Transaktion (abort)**
  - **Alle** Effekte einer Transaktion müssen **rückgängig** gemacht werden, wenn diese abgebrochen wird.
  - Dazu müssen bei **schreibenden** Zugriffe entweder
    - alte Daten gesichert werden oder
    - auf einer Kopie des Speicherbereiches gearbeitet werden ← meistens

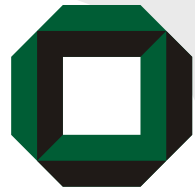
So kann im Falle des Abbruchs der Zustand **vor der Transaktion** wiederhergestellt werden.
- Während einer Transaktion dürfen keine E/A-Operationen ausgeführt werden, die nicht rückgängig gemacht werden könnten.
  - Sog. „Output-Commit“-Problem kann durch Speichern und Wiederherstellen der Eingaben bzw. Zwischenspeichern und Verzögern der Ausgaben behoben werden.



# Transaktionaler Speicher: Konzepte (6)

- **Granularität**

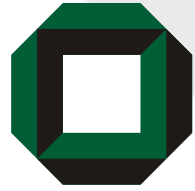
- Bei der Implementierung eines Transaktionalen Speicher-Systems muss festgelegt werden, mit welcher **Granularität** gearbeitet wird.
- Frühere Vorschläge für hard- und softwarebasierte Systeme arbeiteten auf **Worten**, d.h. jede Transaktion bestand aus dem Zugriff auf ein oder mehrere Worte
  - großer Verwaltungsaufwand,
  - großer Speicherbedarf
- Moderne hardwarebasierte Systeme arbeiten mit **Blöcken fester Größe**, insbesondere **Seiten** oder **Cache-Zeilen**.
  - Andere Blockgrößen können unterstützt werden, je nach Fähigkeit der Hardware, die alle Schreib- und Lesezugriffe überwachen muss.
- Moderne STM-Systeme arbeiten dagegen überwiegend **objektorientiert**, um den Aufwand für die Zugriffskontrolle zu minimieren.



# Transaktionaler Speicher: Konzepte (7)

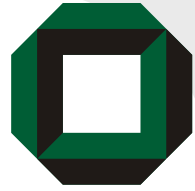
- **Funktionsweise**

- Innerhalb einer Transaktion muss **Buch** geführt werden über alle **Lese-** und **Schreibzugriffe** auf gemeinsam genutzte Speicherbereiche.
- In typischen STM-Systemen werden gemeinsam genutzte Speicherbereiche **deklariert** (durch `open( )`)
  - Dabei teilt der Programmierer dem STM mit, ob er nur lesend, oder auch schreibend auf diese Speicherbereiche zugreifen will.
  - Von Speicherbereichen, auf die geschrieben werden soll, wird eine **Kopie** angelegt, und nur die Kopie modifiziert.
  - Falls die Transaktion erfolgreich abgeschlossen wird, wird mit einer **atomaren Operation** der alte Speicherbereich durch den neuen ersetzt.



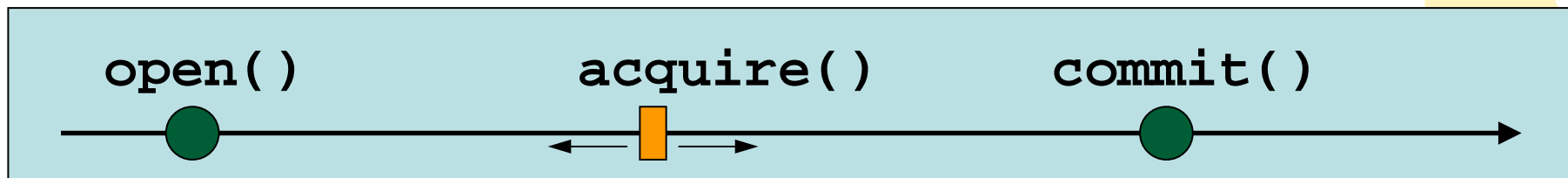
# Konzepte - Konfliktdetektion (1)

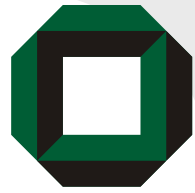
- In einem typischen STM wird die Absicht eines Schreibzugriffs publiziert (d.h. für alle anderen Prozesse sichtbar). Man sagt, dass der jeweilige Prozess versucht, den entsprechenden Block in Besitz zu nehmen (`acquire()`)
- An dieser Stelle werden Konflikte sichtbar:
  - Wenn eine andere Transaktion den fraglichen Bereich in Besitz hält, kann eine der beiden Transaktionen abgebrochen werden.
  - Andere Transaktionen können abgebrochen werden, wenn sie die Daten gelesen haben, für die hier grade der Besitz angefordert wird.
  - Die Transaktion kann abgebrochen werden, wenn Daten ihrer vorhergehenden Lesezugriffe inzwischen überschrieben wurden.



## Konzepte - Konfliktdetektion (2)

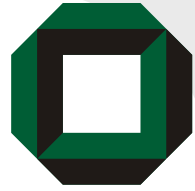
- Die Bekanntgabe der Schreiber ist grundsätzlich unerlässlich für die Konflikterkennung und findet immer vor dem abschließenden, tatsächlichen Festschreiben der Änderungen (Commit) statt.
- **acquire()** ist normalerweise keine eigenständige Operation, sondern Bestandteil von **open()** oder **commit()** (je nach STM-Implementierung). Beides hat Vor- und Nachteile.
- **Frühzeitige Besitznahme:**
  - + Kann Konflikte frühzeitig detektieren.
  - Kann Transaktionen unnötigerweise abbrechen. (z.B. wenn die störende Transaktion selbst später abbricht)
- **Späte Besitznahme:**
  - Lässt zum Scheitern verurteilte Transaktionen lange laufen.
  - + Übersieht Konflikte, die sich später gar nicht auswirken.





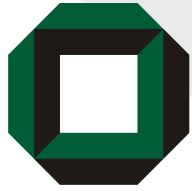
# Implementierungen

- Zunächst als **Hardware-Mechanismus** vorgeschlagen.
  - [HeMo93] Herlihy and Moos: „Architectural Support for Lock-Free Data Structures“, 1992
- Heute überwiegend in **Software** realisiert, z.T. mit Hardware-Unterstützung.
  - [ShTo95] STM: Shavit and Touitou: „Software Transactional Memory“, 1995



# STM-Beispiel-Implementierung (1)

- Im folgenden wird beispielhaft eine STM-Implementierung mit **frühzeitiger** Konflikterkennung gezeigt [He2+03].
  - Prototyp u.A. in Java, verwendet compare-and-swap
- **Transaktion** durch **TMThread**-Klasse realisiert, die von **Thread** erbt und zusätzliche Methoden für Transaktion bereitstellt: `beginTransaction`, `open`, `commit`, `abort`, `checkStatus`, ...
- **Transaktions-Objekt** **TMObject** als Hüll-Klasse realisiert, die Java „Object“ enthält; implementiert `clone`-Operation.



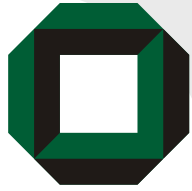
## STM-Beispiel-Implementierung (2)

### Beispiel atomarer Zähler:

```
//Initialisierung
Counter counter = new Counter(0);
TMObject tmObject = new TMObject(counter);

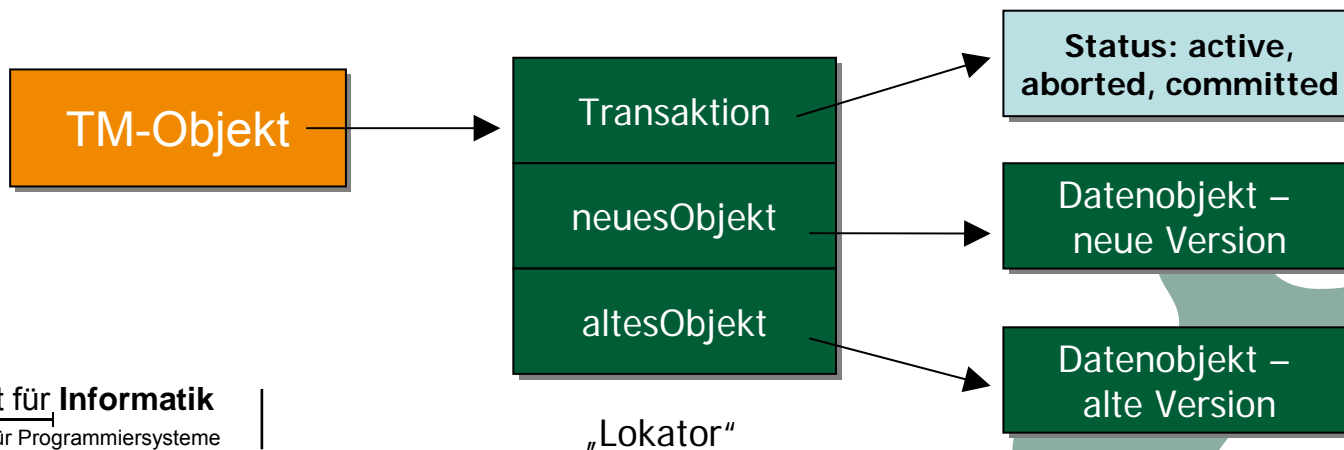
//Verwendung innerhalb einer Transaktion
//Faden ruft beginTransaction auf
Counter counter=(Counter)tmObject.open(WRITE);
counter.inc();
...
```

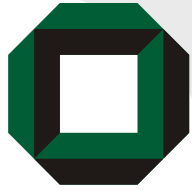
- **open** legt Datenstrukturen für Transaktion einschließlich einer Kopie an (vgl. nächste Folien)
- Faden manipuliert Kopie des Objektes wie gewöhnlich
- Implementierung garantiert, dass kein anderer Faden Zugriff auf lokale Kopie hat
- Transaktion kann von sich aus abbrechen (**abortTransaction**)
- **commitTransaction**: Erzeugt mit CAS eine konsistente neue Version oder schlägt fehl



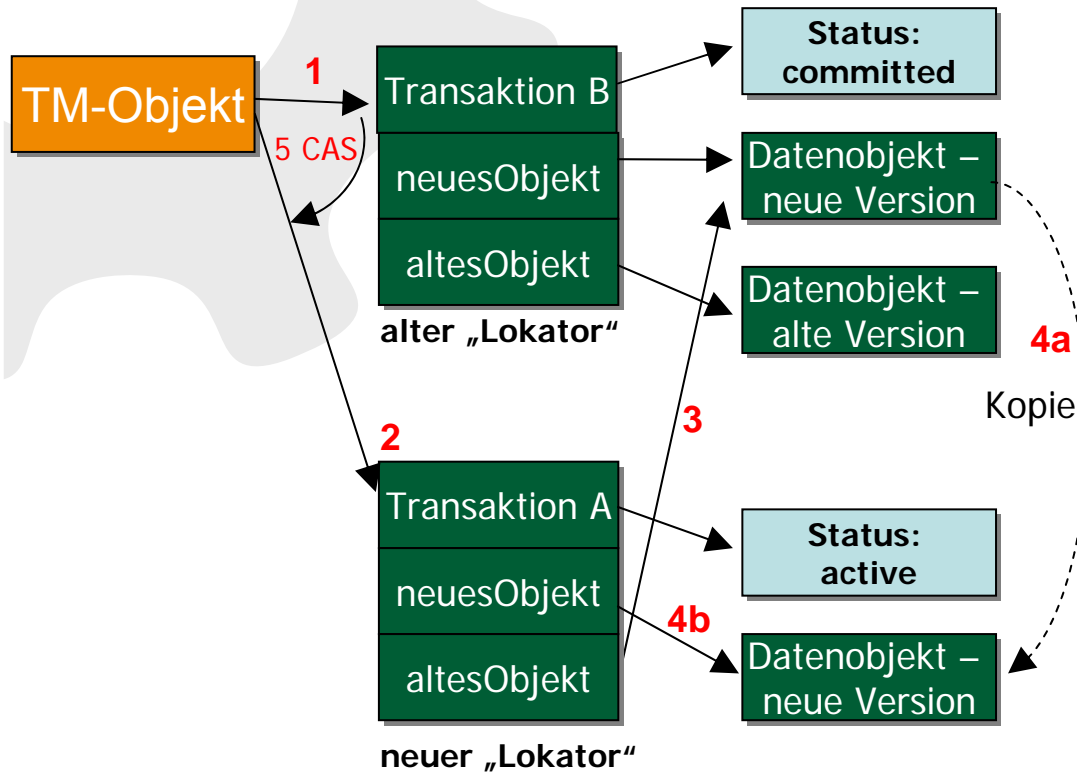
## STM-Beispiel-Implementierung (3)

- Details zur Realisierung eines Transaktions-Objekts
- Jedes Objekt hat 3 Felder (referenziert über „Lokator“)
  - eine **Referenz** auf die letzte Transaktion, die das Objekt im Schreibmodus geöffnet hat
  - **altesObjekt**: Zeiger auf die alte Version des Objekts (vor Beginn der Transaktion)
  - **neuesObjekt**: Zeiger auf neue Version des Objekts (aktuelle Daten). Diese wird von der Transaktion genutzt und ggf. verändert.
- Indirektion über **TM-Objekt** erlaubt atomaren Zugang zu allen Lokator-Feldern.  
Methode: Zeiger mit compare-and-swap auf anderen Lokator „umschwenken“





# STM-Beispiel-Implementierung (4)

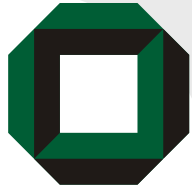


Öffnen eines TMOBJekt in Schreibmodus und **Commit**:

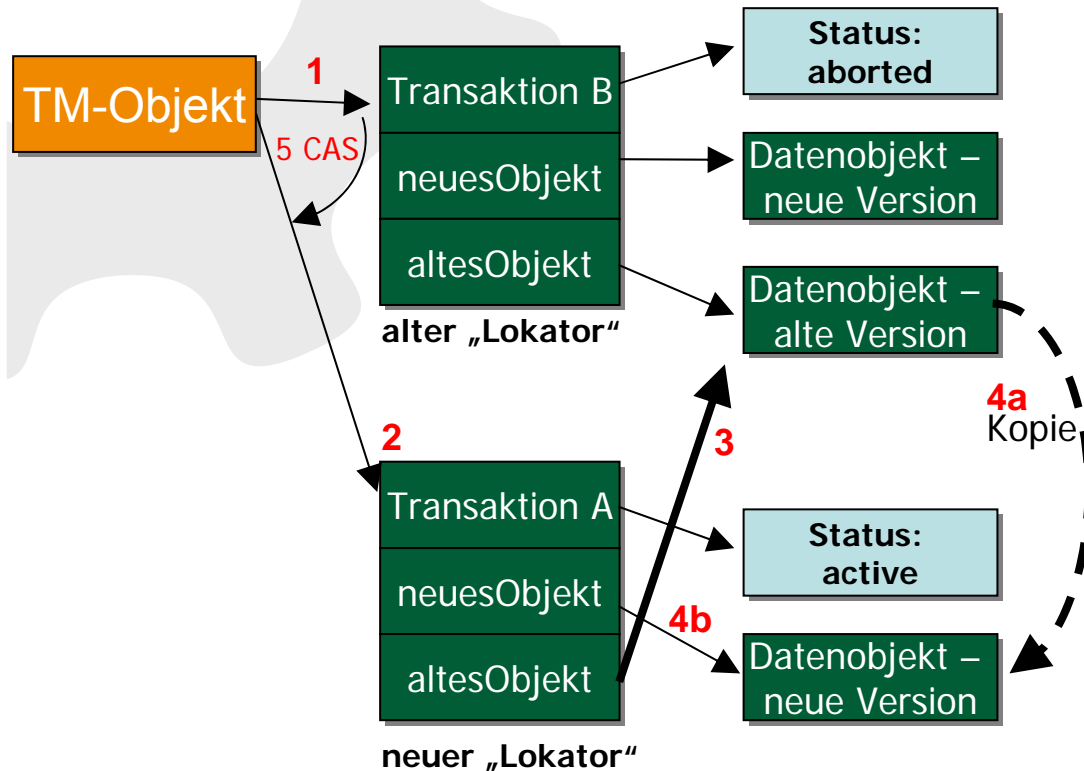
1. Ein Zeiger auf den alten Locator wird gespeichert.
2. Es wird ein neuer Locator angelegt, der auf die neue Transaktion verweist. Deren Zustand wurde mit „active“ initialisiert.
3. Der Zeiger „alt“ des neuen Locators zeigt auf die aktuellen („neu“ nach commit) Daten der abgeschlossenen Transaktion.
4. Der Zeiger „neu“ im neuen Locator zeigt auf eine frisch angelegte Kopie dieser Daten.
5. Mittels einer CAS-Operation wird die global sichtbare Referenz für das gemeinsam genutzte Objekt auf den neuen Locator umgesetzt, wenn die Referenz sich seit (1.) nicht geändert hat (durch Transaktion C).

Annahme:

B abgeschlossen mit commit; A beginnt;  
C (nicht gezeigt) konkurriert mit A



# STM-Beispiel-Implementierung (5)



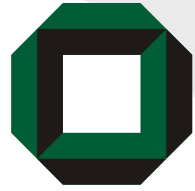
Beispiel nach **Abort** von B:

1. Ein Zeiger auf den alten Locator wird gespeichert.
2. Es wird ein neuer Locator angelegt, der auf die neue Transaktion verweist. Deren Zustand wurde mit „active“ initialisiert.
3. Der Zeiger „alt“ des neuen Locators zeigt auf die „alten“, **ungeänderten** Daten der abgebrochenen Transaktion.
4. Der Zeiger „neu“ des neuen Locators zeigt auf eine frisch angelegte Kopie dieser alten Daten.
5. Mittels einer CAS-Operation wird die global sichtbare Referenz für das gemeinsam genutzte Objekt auf den neuen Locator umgesetzt, wenn sie sich seit (1.) nicht geändert hat (durch Transaktion C).

Annahme:

B abgeschlossen mit abort; A beginnt;

C (nicht gezeigt) konkurriert mit A

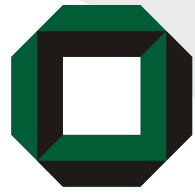


## STM-Beispiel-Implementierung (6)

- Nur lesende Transaktionen
  - Signalisieren „nur Lesen“ durch `tmObject.open(READ);`
  - Am Ende einer solchen Transaktion kann wieder mit

`compare-and-swap (tmObjectRef, alter Verweis, alter Verweis)`

validiert werden, ob die gelesenen Daten noch gültig sind (und nicht von einer anderen Transaktion verändert wurden).



# Probleme mit STM

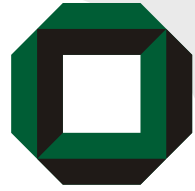
- Programmierfehler immer noch möglich. Beispiel für Verklemmung (Deadlock) [Bl+06]:

```
bool a = false;
bool b = false;

//Thread 1:
//a lesend, b schreibend
atomic {
    while (!a);
    b = true;
}

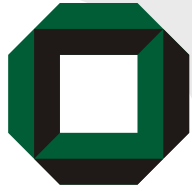
//Thread2:
//a schreibend, b lesend
atomic {
    a = true;
    while (!b);
}
```

- Oft Annahme: „Kurze“ kritische Sektionen [HeMo93]



# Ausblick

- Gegenstand aktueller Forschung:
  - Verbesserung der **Performanz**
  - **Integration** in bestehende Programmiersprachen (Bibliotheken & Spracherweiterungen)
  - Formale Fragestellungen (Vermeidung von Deadlocks, Livelocks, etc.)



# Literatur / Links

- [Bl+06] Colin Blundell et al., Subleties of Transactional Memory Atomicity Semantics, Computer Architecture Letters, Volume 5, Number 2, November 2006
- [Her91] Herlihy, M., Wait-free synchronization, ACM TOPLAS, 1991, 13, 124-149
- [He+03] Herlihy, M. et al., Obstruction-Free Synchronization: Double-Ended Queues as an Example, Proc. ICDS, 2003
- [He2+03] Herlihy, M. et al., Software transactional memory for dynamic-sized data structures, ACM PODC '03, 2003, 92-101
- [HeMo93] Herlihy, M., Moss, J. E. B. Transactional memory: architectural support for lock-free data structures, Proc. ISCA '93, 1993, pp. 289-300
- [LaRa07] Larus, J.R., Rajwar, R., Transactional Memory, Morgan & Claypool Publishers, 2007
- [ShTo97] Shavit, N., Touitou, D. Software transactional memory Distributed Computing, 1997, V10, pp. 99-116
- [Intel07] Intel® C++ STM Compiler, Prototype Edition, <http://softwarecommunity.intel.com/articles/eng/1498.htm> (09/2007)