HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Parallele Programmierung
Transactional Memory

Vorlesung 14

Prof. Dr. Luc Bläser

Inhalt Heute

Transactional Memory

- Konzept und Anwendung

Lernziele

- Transactional Memory kennen lernen
- Nutzen und Limitation des Modells beurteilen

Motivation

- Problem der Shared Mutable Memory
  - Explizite Synchronisation: Deadlocks, Starvation, Kosten
  - Race Conditions bei ungenügender Synchronisation
- Transactional Memory
  - Idee aus Datenbanksystem
  - Ziele: Keine Races, keine Deadlocks, (keine Starvation)
- Umsetzung
  - In SW => Software Transactional Memory (STM)
  - In HW => Hardware Transactional Memory (HTM)

Historisches

- Alte Idee aus Datenbankwelt
- Tom Knight: 1986
  - Hardware Support für Transaktionen
- Maurice Herlihy, J. Eliot B. Moss: 1992
  - Populärer gemacht
- Nir Shavit, Dan Touitou: 1995
  - Reine Software Transaktionen
- 2013 wieder auch in HW
  - Intel Transactional Synchronization Extensions (TSX)

Software Transactions

Atomare Sequenz von Operationen

- Read/Write auf Speicher
- Konzeptionell wie eine grosse atomare Instruktion
- keine (inkonsistenten) Zwischenstände bemerkbar

```
atomic {
    success = balance >= amount;
    if (success) {
        balance -= amount;
    }
}
```

ACI Transaktionen

Atomicity

- Vollständig oder gar nicht sichtbar

Consistency

- Programm vor und nach Transaktion gültig

Isolation

- Effekte wie eine serielle Ausführung

Gegensatz zu DB

- Keine Durability/Persistenz

Software Transactions: Konzept

Deskriptive Programmierung

- Was ist atomar? Nicht wie!

Automatische Isolation

- Überlasse korrekte Ausführung dem System

Einschränkungen

- Nur Speicherzugriffe sind isoliert, Seiteneffekte nicht

Implementierung

- Meist optimistisches Concurrency Control

Transactions versus Locking: Konzept

Software Transactions

```
void deposit(int amount) {
  atomic {
    balance += amount;
  }
}

boolean withdraw(int amount) {
  atomic {
    if (balance >= amount) {
      balance -= amount;
      return true;
    } else {
      return false;
    }
  }
}
```

Deskriptiv:
«Atomar und isoliert»

Locking

```
void deposit(int amount) {
  synchronized(this) {
    balance += amount;
  }
}

boolean withdraw(int amount) {
  synchronized(this) {
   if (balance >= amount) {
      balance -= amount;
      return true;
    } else {
      return false;
    }
  }
}
```

Imperativ:
«Monitor Lock & Unlock»

Nested Transactions

```
void transfer(Account from, Account to, int amount)
```

STM

```
atomic {
  if (from.withdraw(amount)) {
    to.deposit(amount);
  }
}
```

Geschachtelte Transaktionen

- Atomar
- Konsistent (Geldsumme invariant)

Locking

```
if (from.withdraw(amount)) {
  to.deposit(amount)
}
```

Geld fehlt zwischen Withdraw/Deposit

- Nicht atomar
- Inkonsistenz zeitweise sichtbar

Keine Races, keine Deadlock

```
void transfer(Account from, Account to, int amount)
```

STM

```
atomic {
  if (from.withdraw(amount)) {
     to.deposit(amount);
  }
}
```

- Deadlock-Frei
- Auch im Fehlerfall atomar

Locking

```
synchronized(from) {
  if (from.withdraw(amount)) {
    to.deposit(amount);
  }
}
```

Niemand sieht Stand zwischen Überweisung

- Deadlock-Gefahr
- Geld im Fehlerfall verlierbar

Transaktionsausführung

- Meist optimistisches Concurrency Control (OCC)
  - Unsynchronisiert ausführen
  - Bei Konflikten => Rollback durch System
- Transaktionen können unerwartet abbrechen
  - Automatisches Retry solcher Transaktionen
  - Sollte für Anwendung unbemerkbar sein

Typische Probleme

- Seiteneffekte in SW-Transaktionen bleiben sichtbar
  - Sehe Abbruch und Wiederholung
  - Kein IO: Konsole, Log, Files, Netzwerk, UI etc.
- Starvation-Gefahr bei OCC
  - Transaktion u.U. wiederholt wegen Konflikt abbrechbar

Warten auf Bedingungen

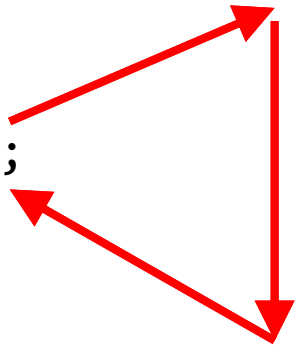
- Retry innerhalb Transaktionen
  - Transaktion ausdrücklich abbrechen
  - Automatische Wiederholung später
- System Monitoring
  - Wiederholung erst, wenn Zustandsänderung
  - Keine explizite Notifizierung

```
atomic {
    if (balance < amount) {
      retry;
    }
    balance -= amount;
}
```

Nested Transactions

- Transaktionen innerhalb Transaktion
- Commit erst bei Top-Level Transaktion

```
atomic {
  from.withdraw(amount);
  to.deposit(amount);
}
```



```
void withdraw(int amount) {
  atomic {
    if (balance < amount) {
      retry;
    }
    balance -= amount;
  }
}
```

Software Transactional Memory (STM)

- **ScalaSTM: Scala und Java**
- Intel C++ STM

Viele ältere Versionen

- Multiverse STM: Java
- Akka STM => ScalaSTM
- Deuce JVM: Spezielle JVM
- SXM: für .NET (MSR)

Hardware Transactional Memory (HTM)

Intel Transactional Synchronization Extensions (TSX)

- Unterstützt sogar Nested Transactions

Kann von STM Framework genutzt werden

- Hohe Effizienz möglich

```
XBEGIN fallback
  Instructions
XEND

fallback: // bei Abbruch
```

STM auf Java

Beispiele für Scala STM mit Java

- Keine spezielle JVM, dafür Kompromisse

Prinzip ähnlich zu anderen STM Systemen

- Clojure, Multiverse STM etc.

Wrapping von Variablen

Wrapper für transaktionelle Zugriffe

- Müsste sonst auch JVM-Ebene detektiert werden
- `get()` und `set()` möglich

Keine normalen Variablenzugriffe in Transaktionen

- Sind nicht transaktionell (wie sonstige Seiteneffekte)
- Ausnahme: Lokale Variablen/Parameter

```
final Ref.View<Integer> balance = STM.newRef(0);
final Ref.View<LocalDate> lastUpdate = STM.newRef(LocalDate.now());
```

Transaktion

```
import scala.concurrent.stm.japi.STM;

void deposit(int amount) {
  STM.atomic(() -> {
    balance.set(balance.get() + amount);
    lastUpdate.set(LocalDate.now());
  });
}
```

Retry

```
void withdraw(int amount) {
  STM.atomic(() -> {
    if (balance.get() < amount) {
      STM.retry();
    }
    balance.set(balance.get() - amount);
    lastUpdate.set(LocalDate.now());
  });
}
```

Nested Transactions

```
void transfer(BankAccount from,
              BankAccount to, int amount) {
  STM.atomic(() -> {
    from.withdraw(amount);
    to.deposit(amount);
  });
}
```

Zugriffe in Transaktionen

Nur Zugriffe auf `Ref.View<T>` Wrapper

- T muss immutable sein (sonst auch nicht atomar)

Indirekte Strukturen sind nicht transaktionell

- `Ref.View<Map<K, V>>`: Map ist nicht transaktionell
- Vordefinierte transaktionelle Collections
  - `STM.newMap()`
  - `STM.newSet()`
  - `STM.newArrayAsList()` – fixed sized list
  - Normale List fehlt

Transactional Collection

```
final Map<String, BankAccount> accounts = STM.newMap();

BankAccount openAccount(String name) {
  return STM.atomic(() -> {
    BankAccount account = new BankAccount();
    accounts.put(name, account);
    return account;
  });
}
```

Keine Seiteneffekte in Transaktionen

Generell: Kein IO

- Konsole, Files, Netzwerk, Thread Start/Wait/Join

Spezifisch: Keine normalen Variablen

- Nur Lesen von final Variablen oder lokalen Variablen
- Sonst nur `Ref.View<T>` Zugriffe mit immutable T
- Oder Transactional Collections

Transaktionstheorie

- Jede serielle Ausführung der Transaktionen ist korrekt
- Nebenläufige Ausführung auch korrekt, sofern die Effekte gleich wie irgendeine serielle Ausführung sind

Begriff der Serialisierbarkeit

→ Siehe Datenbankvorlesung DB1

Transaction Memory: Diskussion

Deskriptiver Ansatz

- Einfaches Programmiermodell
- Keine Race Conditions und Deadlocks

Komplexe Implementierung

- Hohe Laufzeit- und Speicherkosten
- Starvation-Vermeidung

Diverse Schwächen in Frameworks

- Seiteneffekte, Starvation, Write Skews, Ref-Wrapper

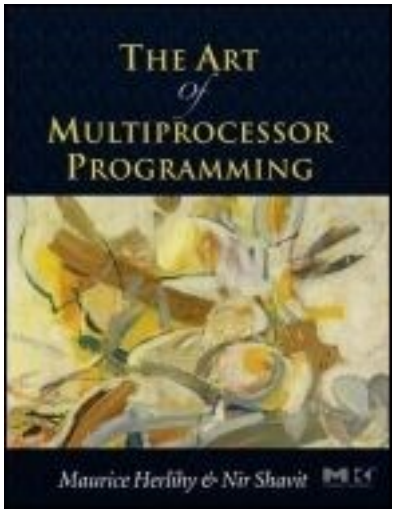
Lernziele Rückblick

- Transactional Memory kennen lernen
- Nutzen und Limitation des Modells beurteilen

Literatur (Selbststudium)

M. Herlihy and N. Shavit. The Art of Multiprocessor Programming, Revised Reprint, Morgan, 2012

- Kapitel 18: Transactional Memory



Scala STM

- In Scala, aber auch in Java benutzbar
- https://nbronson.github.io/scala-stm/quick_start.html

HTM Technologie

Intel Transactional Synchronization Extensions (TSX)

- https://software.intel.com/sites/default/files/m/9/2/3/41604 (Kapitel 8)
- Prototyp mit STM Sprachelemente für C/C++ https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition