Software Transactional Memory

(Software-Architektur)

Prof. Dr. Oliver Braun

Letzte Änderung: 11.07.2017 15:12

Transaktionaler Speicher

- um nebenläufige Programme schreiben zu können, aber Blockierungen zu vermeiden
- Speicherzugriff in Transaktionen
- Idee aus dem Bereich der Datenbanken
- bisher hauptsächlich **Software Transactional Memory**
- in Hardware, z.B.
  - Intel: Transactional Synchronization Extensions
  - SUN, 2008: 16-kernigen Sparc-Prozessors mit Hardware-unterstütztem transaktionalem Speicher (2010 nach Übernahme durch Oracle eingestellt)
- Maurice Herlihy, J. Eliot B. Moss: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th International Symposium on Computer Architecture (ISCA). 1993, S. 289–300

STM

- Mediator zwischen kritischen Sektionen von Code (atomarer Block) und der Heap
- STM greift bei Lese- und Schreibzugriffen ein und überwacht die Zugriffe
- wenn die Zugriffe überlappen, werden alle zurückgerollt (*roll back*) und erneut versucht (*retry*)
- ansonsten wird der atomare Block *committed*
- andere Threads sehen nur committete Zustände

Optimistic Concurrency Control

- STM nutzt optimistische Nebenläufigkeitskontrolle
- STM nimmt an, dass atomare Blöcke gleichzeitig ablaufen können
- wenn die Annahme nicht korrekt ist, wird der Anfangszustand wieder hergestellt und es erneut versucht
- den alten Zustand aufzuheben bedeutet natürlich etwas Overhead
- optimistische Ansätze skalieren aber üblicherweise besser

ScalaSTM

- es gibt zahlreich, ambitionierte Ansätze die versuchen mit STM sequentiellen, imperativen Code parallel auszuführen
- dieser schwierige Ansatz erfordert einiges an Magie, weil für jeden Lese- und Schreibzugriff von veränderlichen Variablen STM-Aufrufe eingefügt werden müssen
- dadurch ist eine gute Performanz auch schwer zu erzielen
- ScalaSTM, wie auch Haskell- und Clojure-Implementierungen, verwaltet explizite `Refs`
- es sind keine Bytecode-Manipulationen oder Compiler-Modifikationen notwendig
- ScalaSTM ist als Bibliothek implementiert
- auch Concurrent Sets und Maps können in Transaktionen genutzt werden

Vorteile

- *Say what you mean*. Ein Block muss nur mit `atomic` gekennzeichnet werden. Atomare Blöcke können verschachtelt werden.
- Lesezugriffe skalieren. Alle Threads können lesend zugreifen.
- Exceptions triggern automatisches Cleanup.
- Auf komplexe Bedingungen zu warten, ist einfach. Wenn ein atomarer Block nicht den erwarteten Zustand vorfindet, kann er `retry` aufrufen und auf Änderungen der Inputs warten. Verschiedene Lösungswege können einfach verkettet werden.
- Einfach. ScalaSTM ist ein JAR ohne Abhängigkeiten.

Nachteile

- Jeder Zugriff (lesend oder schreibend), braucht `()`. Wenn `x` eine `Ref` ist, kann sie mit `x()` gelesen und mit `x() = y` geschrieben werden.
- Single-Thread-Overhead.
- Rollback funktioniert (natürlich) nicht mit I/O. `Refs` werden korrekt zurück gesetzt, aber Ausgaben o.ä. können natürlich nicht rückgängig gemacht werden.

QuickStart

- siehe auch https://nbronson.github.io/scala-stm/quick_start.html
- es soll eine veränderbare, doppelt verkettete Liste von mehreren Threads gemeinsam genutzt werden
- um ScalaSTM in einem Sbt-Projekt zu nutzen, reicht die folgende Zeile in der Datei `build.sbt`:

```
libraryDependencies += ("org.scala-stm" %% "scala-stm"
```

Ref für Shared Variables

```
import scala.concurrent.stm._

class ConcurrentIntList {
  private class Node(val elem: Int, prev0: Node, next0: Noc
    val isHeader = prev0 == null
    val prev = Ref(if (isHeader) this else prev0)
    val next = Ref(if (isHeader) this else next0)
  }

  private val header = new Node(-1, null, null)
```

- Zugriff auf die Pointer `prev` und `next` soll threadsafe sein
- beim Erzeugen eines Knoten zeigen die beiden Pointer auf den Knoten selbst
- die beiden Pointer sind nie `null`
- die Liste beginnt und endet mit einem Pseudoknoten `header`

Zugriff in atomic wrappen

```
def addLast(elem: Int) {
  atomic { implicit txn =>
    val p = header.prev()
    val newNode = new Node(elem, p, header)
    p.next() = newNode
    header.prev() = newNode
  }
}
```

- lesende und schreibende Zugriffe auf den Inhalt einer `Ref` können nur in einem `atomic`-Block erfolgen
- Trick um das vom Compiler checken zu lassen:
  - ein impliziter `InTxn`-Wert muss vorhanden sein
  - aus diesem Grund beginnt der Block mit

    ```
    atomic { implicit txn =>
    ```

atomic-Blöcke komponieren

- atomic-Blöcke können beliebig verschachtelt werden
- z.B.

```
def addLast(e1: Int, e2: Int, elems: Int*) {
  atomic { implicit txn =>
    addLast(e1)
    addLast(e2)
    elems foreach { addLast(_) }
  }
}
```

Warten auf veränderte Bedingungen

```
def removeFirst(): Int = atomic { implicit txn =>
  val n = header.next()
  if (n == header)
    retry
  val nn = n.next()
  header.next() = nn
  nn.prev() = header
  n.elem
}
```

- `retry` macht ein Rollback
- und wartet bis ein anderer Thread eine Änderung an einer der `Refs` gemacht hat, die in der Transaktion gelesen werden
- erst dann wird die Transaktion erneut gestartet
- `removeFirst` wartet solange bis die Liste nicht leer ist

Alternative statt `retry`

```
def maybeRemoveFirst(): Option[Int] = {
  atomic { implicit txn =>
    Some(removeFirst())
  } orAtomic { implicit txn =>
    None
  }
}
```

- wenn `removeFirst` `retry` aufruft, wird die Kontrolle von `maybeRemoveFirst` übernommen
- statt dem `retry` wird dann der zweite Block ausgeführt und `None` zurück gegeben
- das blockierende Verhalten von `removeFirst` wird also von **außen** geändert

... und umgekehrt

```
object ConcurrentIntList {
  def select(stacks: ConcurrentIntList*):
        (ConcurrentIntList, Int) = {
    atomic { implicit txn =>
      for (s <- stacks) {
        s.maybeRemoveFirst() match {
          case Some(e) => return (s -> e)
          case None => _
        }
      }
      retry
    }
  }
```

- `select` blockiert solange bis es ein Element von einer der übergebenen Listen entfernen kann

Dijkstras Philosphenproblem mit ScalaSTM

```
class Fork {
  val owner = Ref(None : Option[String])
}

class PhilosopherThread(val name: String, val meals: Int,
        left: Fork, right: Fork) extends Thread {
  val mealsEaten = Ref(0)

  override def run() {
    for (m <- 0 until meals) {
      // thinking
      atomic { implicit txn =>
        if (!(left.owner().isEmpty && right.owner().isEmpty
          retry
        left.owner() = Some(name)
        right.owner() = Some(name)
      }
      // eating
```