STM: Compiler-Integration und Sprachen-Erweiterungen

Andreas Zwinkau

IPD, Lehrstuhl Prof. Tichy

Zusammenfassung. Es existieren eine Anzahl von Implementierungen von „Software Transactional Memory“ (STM). Diese Ausarbeitung gibt einen Einblick in die Probleme und internen Funktionsweisen von verschiedenen STM-Systemen.
Zwei Beispielprobleme, Seiteneffekte und Barrierefreiheit, zeigen wo die Schwierigkeiten liegen. Eine Javabibliothek demonstriert beispielhaft, wie man STM mit vorhandenen Sprachkonstrukten einsetzen kann und C++ Bibliotheken beweisen die Leistungsfähigkeit solcher Systeme. Wie STM sprachlich integriert werden kann, sieht man bei einem modifzierter Javaübersetzer und der Sprache ATOMOΣ . In diesem Zusammenhand wird auch Optimierungsmöglichkeiten eingegangen. Abschließend zeigt Haskell wie STM in einer funktionalen Sprache realisiert werden kann.

1 Einführung

Prozessorhersteller konzentrieren sich immer mehr auf Mehrkernarchitekturen. Um die Leistungskraft paralleler Rechner voll ausnutzen zu können, müssen Programme parallel arbeiten und synchronisiert werden. Herkömmliche Synchronisiertechniken wie Sperren[1] oder Semaphoren sind allerdings schwierig zu benutzen und häufige Ursache von Fehlern.

Das Konzept des Transaktionalen Speichers ist einfacher und sicherer. Software Transactional Memory (STM), also eine Implementierung ohne spezielle Hardwareunterstützung, wurde bereits in verschiedenen Varianten entworfen und ausgewertet. Dieser Artikel gibt einen Überblick über einige konkrete Ansätze. Ich gehe davon aus, dass der Leser mit dem allgemeinen Konzept von Software Transactional Memory vertraut ist.

In Kapitel 2 werden zwei Beispielprobleme gezeigt, die bei der Implementierung auftauchen. Kapitel 3 behandelt Bibliotheken, die STM implementieren. Kapitel 4 zeigt wie mit Spracherweiterungen STM syntaktisch in Programmiersprachen integriert wird.

2 Beispielprobleme

Die Implementation von Transaktionalem Speicher wirft einige Designfragen auf und nicht alle können bisher klar beantwortet werden. Dieses Kapitel stellt beispielhaft zwei Probleme vor.

2.1 Das Blockadeproblem

Ein „blockadefreies“ System garantiert einer Transaktion Fortschritt, falls alle anderen Transaktionen untätig sind. Mit anderen Worten eine Transaktion kann eine andere nicht blockieren.

Als Beispielszenario stelle man sich eine Transaktion L vor, die ein Jahr lang rechnet. Gleichzeitig werden mehrere Transaktionen K_i ausgeführt, die im Konflikt mit L stehen.

Falls das System blockadefrei ist, wird L während seiner Berechnung von einem K_i ungültig gemacht werden. Infolgedessen wird nach einem Jahr, wenn L fertig wird, die Transaktion L fehlschlagen und wiederholt werden. Erst wenn alle K_i, die L stören, abgearbeitet sind wird L erfolgreich sein. Das System als Ganzes macht allerdings garantiert Fortschritt.

[1] engl.: locks

In einem System das nicht blockadefrei ist, kann man andere Transaktionen blockieren. L könnte als die konfliktbehaftete Resource blockieren und ungehindert ausgeführt werden. Die K_i müssen warten, aber es wird keine Rechenzeit verschwendet.

Eine Verklemmung[2] ist in einem STM-System kein Problem, da alle Transaktionen und Ressourcen zentral überwacht werden können.

Ennals [3] zeigt, dass Blockadefreiheit sich negativ auf die Leistung auswirken kann. Die oben beschriebene, sich wiederholende Invalidierung ist ein Grund. Ein anderer sind Cachekonflikte (bzw. Cache Trashing) die auftreten, wenn verschiedene Transaktionen von verschiedenen Prozessoren auf denselben Speicher zugreifen.

Durch einen Vergleich zwischen zwei Implementierungen zeigt Ennals, dass blockadefreie STM-Systeme in manchen Tests bis zu doppelt so lange brauchen, wie nicht-blockadefreie Systeme. Man kann bei diesen Benchmarks große Unterschiede bei Cache und TLB Verfehlungen beobachten.

2.2 Das Seiteneffektproblem

Ein besonderes Problem bei Transaktionen besteht darin, dass diese mehrmals ausgeführt werden könnten, wenn sie nicht gleich beim ersten Mal erfolgreich sein. Was passiert dabei aber mit den Seiteneffekten, wie zum Beispiel Ein- und Ausgabeoperationen?

Ein Lösung für Java STMs schlägt Kapselobjekte[4] vor. Zum Beispiel könnte man einen `AtomicOutputStream` benutzen. Die Transaktion ist in diesem Codebeispiel mit dem `atomic` gekennzeichnet.

Listing 1.1. AtomicOutputStream

```
public class ExampleOutput {
        static PrintStream out =
                new PrintStream(
                        new AtomicOutputStream(System.out));
        static void print_sum(int x, int y) {
                atomic {
                        int result = x + y;
                        out.println("Result is " + result);
                }
        }
}
```

Dieses Vorgehen setzt die Möglichkeit eines Rückrufs[3] voraus. Der `out.println` Aufruf schreibt nicht direkt weiter, sondern puffert die Daten und registriert einen Rückruf in der aktuellen Transaktion. Wenn die Transaktion erfolgreich war, findet der Rückruf statt und die Daten weitergeschrieben.

Noch komplizierter wird das mit Datenbanken, die ja ihre eigenen Transaktionen anbieten. Es können nicht beide Arten von Transaktionen als Letzte laufen. Der Vorschlag ist hier, eine weitere Stufe `prepare` einzufügen.

3 Bibliotheken

Dieses Kapitel zeigt wie STM in bekannten Programmiersprachen realisiert werden kann und vergleicht die Leistung von STM mit den herkömmlichen Methoden.

[2] engl.: deadlock (und livelock)
[3] engl.: callback

3.1 API Beispiel

Herlihy et al entwarfen ein flexibles Rahmenwerk um Transaktionalen Speicher zu implementieren[7]. Die Intention ist es ein Rahmenwerk zur Verfügung zu stellen, das als standardisierte Schnittstelle für verschiedene Implementierungen und Tests dienen kann.

Dieses STM-System ist bewusst sehr generisch gehalten, deswegen auch der Name Dynamic STM (DSTM). Der Ansatz versucht einen „lock-in“ auf eine Implementierung zu vermeiden. Die Autoren argumentieren, dass mit STM noch mehr experimentiert werden sollte, bevor man es fest in Compiler einbaut.

Als Beispiel der API dient eine verkettete Liste. Das Interface einer Node wird mit `atomic` annotiert [4].

Listing 1.2. atomic Interface

```
@atomic public interface Node {
  int getValue ();
  void setValue ( int value );
  Node getNext ();
  void setNext ( Node value );
}
```

Die zugehörige Klasse (`INode` in unserem Beispiel) wird ohne Rücksicht auf parallelen Ausführung entwickelt. Anschließend generiert man eine Fabrik für diese Klassen.

Listing 1.3. factory generation

```
Factory<INode> factory = dstm2.Thread.makeFactory(INode.class);

INode intSet = factory.create();
```

Der Gebrauch der Liste (insert, remove, ...) ist nun fast identisch mit der herkömmlichen Methode, nur dass man immer `factory.create()` benutzen muss, statt mit `new` neue Objekte zu erzeugen.

Wer die transaktionale, verkettete Liste benutzen will, muss die Aufrufe in Transaktionen packen. DSTM realisiert Transaktionen als `Callable` Objekte. Diese Objekte werden dann an `Thread.doit` übergeben.

Listing 1.4. doit

```
result = dstm2.Thread.doit(new Callable<Boolean>() {
        public Boolean call() {
                return intSet.insert(value);
        }
});
```

Die `doit` Methode führt die Transaktion in einer Schleife immer wieder aus, bis sie entweder erfolgreich war oder eine `Exception` wirft.

Listing 1.5. The transaction retry loop

```
public static <T> T doIt(Callable<T> xaction) {
  T result = null ;
  while (! Thread.stop ) {
    beginTransaction ();
    try {
       result = xaction.call ();
```

[4] Java 5 Annotations: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

```
    } catch (AbortedException d) {
    } catch (Exception e) {
      throw new PanicException("Unhandled exception " + e );
    }
    if (commitTransaction ()) {
      return result ;
    }
  }
}
```

Man beachte die Fabriken! Diese müssen unter anderem sicherstellen, dass alle Attribute skalar oder `@atomic` sind. Vor allem aber müssen sie die Kapselklasse erstellen, die in Transaktionen dafür sorgen, dass die Aktionen abgebrochen oder zurückgesetzt werden können.

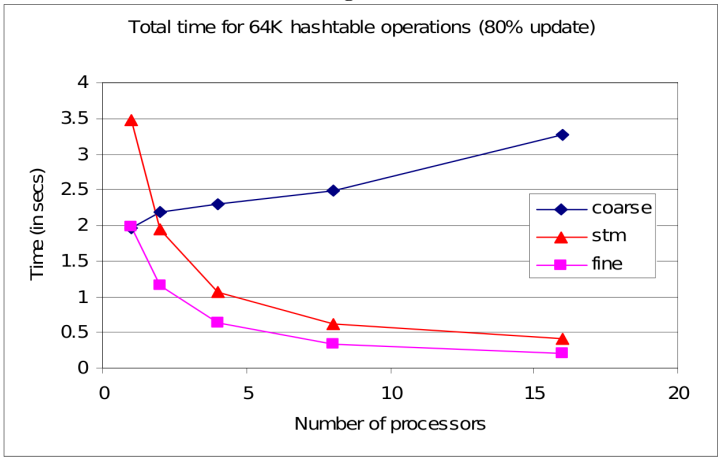
Ein Programmierer kann mit diesem Framework ein eigenes STM-System bauen, falls er mit manchen Designentscheidungen (z.B. Blockadefreiheit weiter oben) nicht einverstanden ist. Er muss dazu eine eigene `Factory` erstellen und `dstm2.Thread.makeFactory` ersetzen.

3.2 Leistung

Transaktionaler Speicher ist eine Abstraktion der Speicherverwaltung. Eine Abstraktion birgt die Gefahr große Leistungsverluste. Aus diesem Grund sind Vergleichstests sehr wichtig.

Multi-Core RunTime (McRT) Als ersten wollen wir dazu die McRT-STM Laufzeitbibliothek [9] betrachten. McRT ist in C/C++ implementiert und getestet werden einige Datenstrukturen (Hashtabelle, Rot-Schwarz-Baum, verkettete Liste). In den Abbildungen 1 und 2 sieht man, dass Transaktionaler Speicher (stm) den herkömmlichen Sperren (locks) teilweise sogar überlegen ist. Es kommt dabei auch auf den konkreten Anwendungsfall an.

Abb. 1. McRT Vergleichstest Hashtabelle



Ein weiterer Test zeigt in Abbildung 3, dass McRT auch mit den Herausforderungen der realen Welt zurecht kommt. Die Filterbibliothek von sendmail wurde modifiziert. Die Messungen ergaben praktisch gleichbleibende Leistung.

Abb. 2. McRT Vergleichstest Verkettete Liste

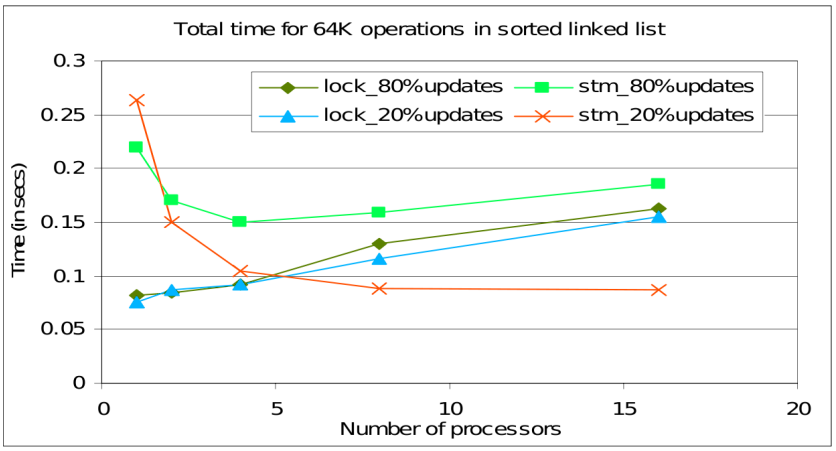
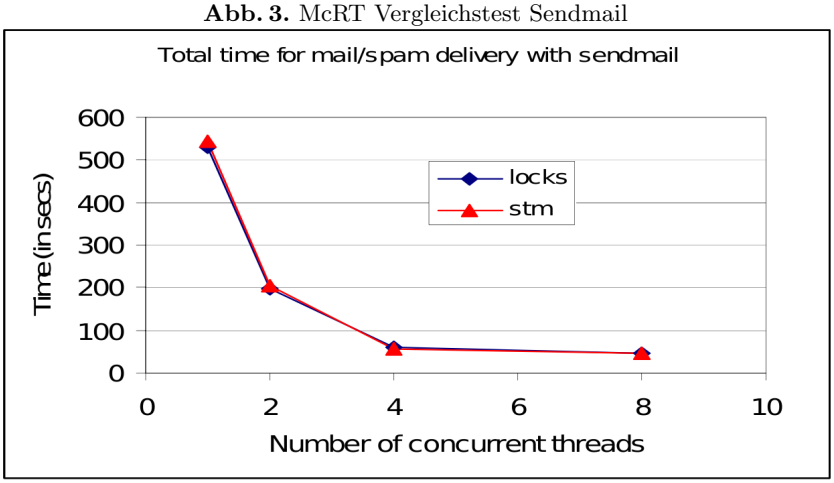


Abb. 3. McRT Vergleichstest Sendmail



Rochester Software Transactional Memory (RSTM) Ein zweites System[8] das wir betrachten, ist ebenfalls in C/C++ implementiert und trotz den oben erwähnten Nachteile blockadefrei.

Das RSTM System wird in verschiedenen Vergeleichstests (Hashtabelle, Rot-Schwarz-Baum, verkettete Liste) mit anderen STM-Systemen und mit Sperren (locks) verglichen. Dabei wird bei den Sperren zusätzlich zwischen grob (coarse-grained) und fein (fine-grained) unterschieden.

Mit „grob“ ist der Einsatz weniger Sperren gemeint, die große Teile des Codes überdecken. Ein „feiner“ Einsatz bedeutet, dass mehrere, kürzere Codeteile gesperrt werden, was mehr Verwaltungskosten verursacht, aber mehr Flexibilität bei der Ablaufkoordination gibt.

RSTM lässt sich wahlweise mit sichtbaren und unsichtbaren Lesezugriffen (visible/invisible reads) und mit eifriger und fauler Aneignung (eager/lazy acquire) der Resourcen benutzen.

Bei der Sichtbarkeit der Lesezugriffe geht es darum, ob die Leser einer Ressource in einer Liste festgehalten werden. Wenn eine Transaktion erfolgreich war und einen Wert verändert, müssen alle Transaktionen, die diesen Wert lesen, neu gestartet werden. Ohne Liste, die Leser *sichtbar* macht, ist es sehr viel aufwendiger, diese zu suchen. Allerding bedeutet eine solche Liste zusätzliche Verwaltungskosten.

Wenn die Aneignung von Objekten bzw. die Konflikterkennung *faul* abläuft, wird Rechenzeit mit Transaktionen vergeudet, die man schon vorher hätte abbrechen können. Eine *eifrige* Konflikt-

erkennung hält aber einer Transaktion den Weg frei, die am Ende doch nicht erfolgreich ist und Rechenzeit wurde verschwendet.

In den Diagrammen sieht man, dass mit vielen Prozessoren RSTM meist schneller arbeitet als Sperren. Die Abbildungen 4,5,6 und 7 zeigen jeweils alle Kombinationen von sichtbar/unsichtbar und eifrig/faul.

Abb. 4. RSTM Vergleichstest Rot-Schwarz-Baum

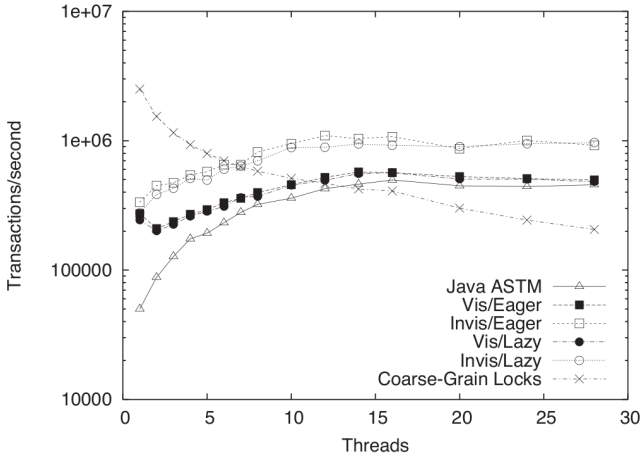
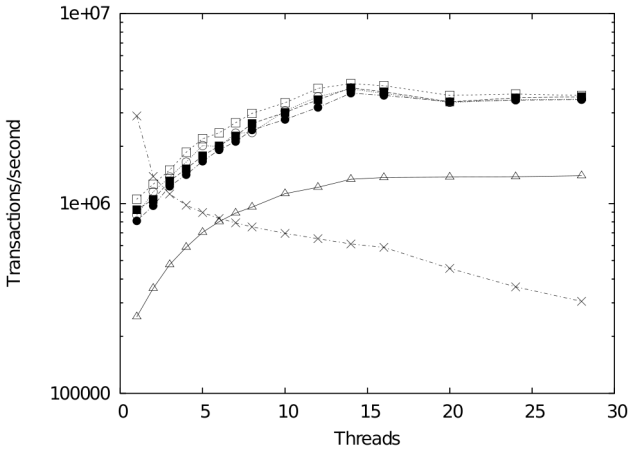


Abb. 5. RSTM Vergleichstest Hashtabelle



Zu den Benchmarks gibt es einige Diagramme, die zeigen, wieviel Zeit RSTM jeweils mit dem Kopieren von Objekten, Konflikt- und Speicherverwaltung, Buchführung, Validierung und Rechnen verbringt. Abbildung 8 zeigt das Diagram zum Rot-Schwarz-Baum Test.

Die Frage ob eifrig oder faule Aneignungen bzw. sichtbare oder unsichtbare Lesezugriffe lässt sich leider nicht klar beantworten. Es hängt vom Verhältnis der Anzahl an Threads zur Anzahl der Prozessoren ab und auch von der Aufgabe.

Abb. 6. RSTM Vergleichstest Verkette Liste

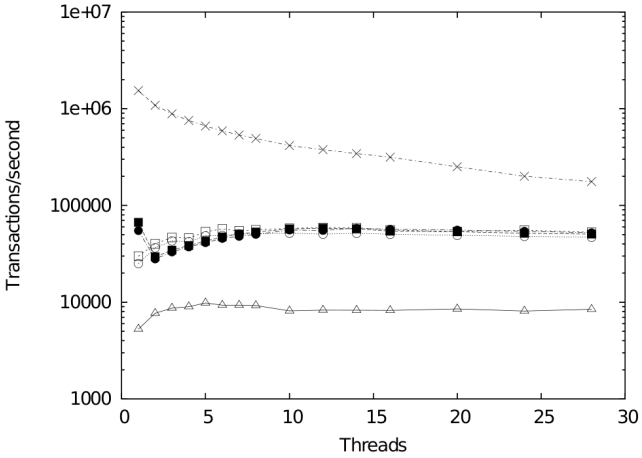


Abb. 7. RSTM Vergleichstest Verkettete Liste mit frühem Loslassen

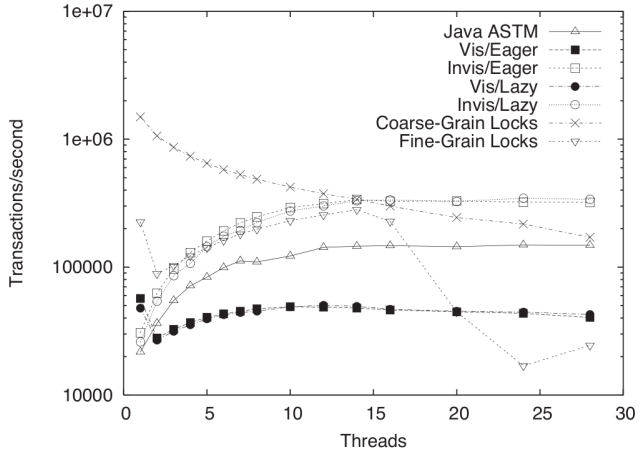
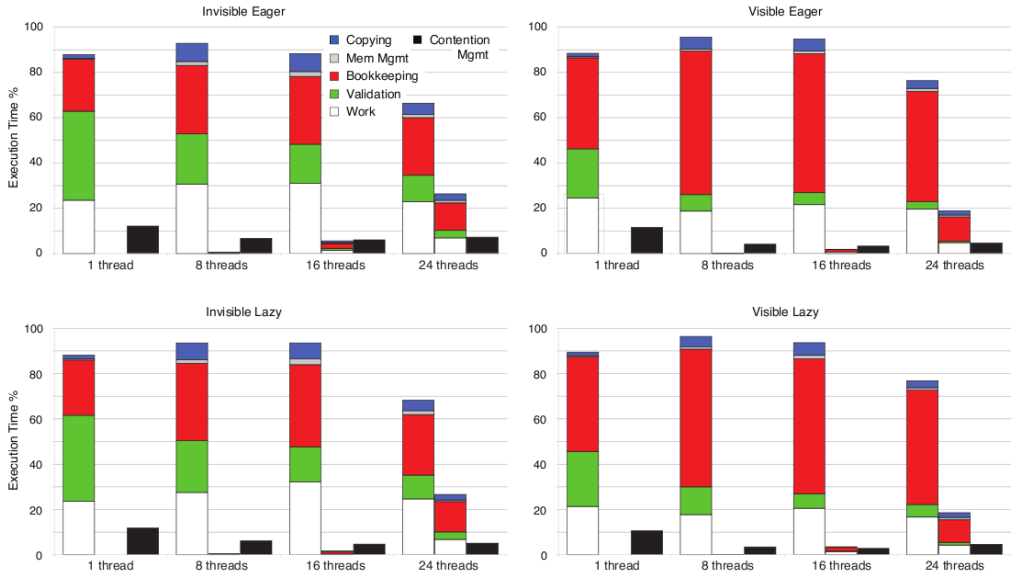


Abb. 8. RSTM Overhead beim Rot-Schwarz-Baum



4 Spracherweiterungen

Nun haben wir gesehen, wie im Kern STM implementiert wird. Der nächste Schritt ist diese Systeme auch in Programmiersprachen einzubinden.

4.1 Von der Bibliothek zur Spracherweiterung

Das erste Beispiel ist eine modifizierte Version von SUNs JVM 1.2.2 von Tim Harris und Keir Fraser [5]. Die Implementierung des Transaktionalen Speichers ist zweigeteilt. Der erste Teil betrifft die Quellcode-zu-Bytecode Schicht. Der zweite Teil sitzt in der Bytecode-zu-nativ Schicht.

Der Bytecode (*.class Dateien) in der Mitte ist unverändert. Die neue Syntax wird in Methodenaufrufe umgewandelt und so in normalen Java Bytecode übersetzbar. Allgemein ist das neue Syntaxkonstrukt folgendermaßen aufgebaut:

Listing 1.6. basic atomic block

```
atomic (condition) {
        statements;
}
```

Der Übersetzer wandelt diesen Code dann in normalen Javacode um. Dieser Vorgang findet mit abstrakten Syntaxbäumen statt. Zurückübersetzt würde der Code so aussehen:

Listing 1.7. basic STM block

```
boolean done = false;
while (!done) {
        STMStart();
        try {
                if (condition) {
                        statements;
                        done = STMCommit();
                } else {
```

```
                        STMWait();
                }
        } catch (Throwable t) {
                done = STMCommit();
                if (done) {
                        throw t;
                }
        }
}
```

Man vergleiche diesen Code mit dem Codebeispiel 1.5.

Im Übersetzer werden noch spezielle Suffixe an die Methodennamen gehängt, um dem Bytecode-zu-nativ Teil zu zeigen, welche Methoden `atomic` sind. Ein `atomic` Block innerhalb einer Methode wird extrahiert in eine eigene Methode.

Diese gekennzeichneten Methoden können dann beim Übersetzen von Bytecode-zu-nativ anders behandelt. Es wird Validierungscode eingfügt, um Schleifen zu vermeiden. Außerdem werden als `volatile` gekennzeichnete Variablen als "kleine", geschachtelte Transaktionen behandelt.

Eine triviale Modifikation für bestehende Javaprogramme mit diesem System wäre `synchronized` einfach durch `atomic` zu ersetzen.

Das zweite Beispiel ist ATOMOΣ [2]. Die vorgeschlagene Syntax für `atomic` Blocks ist praktisch identisch zu der vorhergehenden. ATOMOΣ bietet aber noch zusätzliche Konstrukte an, um dem Programmierer eine feinere Kontrolle zu geben. Beispielhaft sehen wir uns hier `watch` und `retry` am Beispiel einer Barriere[5] an.

Listing 1.8. basic atomic block

```
synchronized (lock) {
        count++;
        if (count != thread_count)
                lock.wait();
        else
                lock.notifyAll(); }
```

In ATOMOΣ schreibt man das so:

Listing 1.9. basic atomic block

```
atomic { count++; }
atomic { if (count != thread_count) {
                watch count;
                retry; }}
```

Die Bedeutung von `watch` ist, dass `count` zur Beobachtungsmenge[6] hinzugefügt wird. Im Anschluss blockiert `retry` den aktuellen Thread solange, bis sich eine Variable in der Beobachtungsmenge ändert - dann wird die Transaktion neu gestartet.

Noch eine Besonderheit bei ATOMOΣ sind verschachtelte Transaktionen. Ein `atomic` innerhalb eines `atomic` verursacht eine Vereinigung der Beobachtungsmenge und Änderungen im inneren Block werden nur für die äußere Transaktion sichtbar.

Manchmal muss man aber eigenständige Transaktionen innerhalb von Transaktionen anwenden. Eine ID Generierung wäre ein Beispiel:

Listing 1.10. ID Generation atomic

```
public int generateID {
        atomic { return id++; }}
```

[5] Alle Threads müssen angekommen sein, bevor sie weiterlaufen dürfen

[6] engl.: watch set

Wenn diese Funktion nun von zwei Transaktionen aus aufgerufen wird, is `id` in beiden Beobachtungsmengen vorhanden und eine von beiden Transaktionen wird fehlschlagen.

ATOMOΣ bietet für solche Fälle das Schlüsselwort `open` an:

Listing 1.11. ID Generation open

```
public int generateID {
        open { return id++; }}
```

Hier werden die Beobachtungsmengen nicht verschmolzen, sondern falls diese kleine Transaktion erfolgreich war, sind die Ergebnisse sofort global verfügbar. D.h. die zweite Transaktion arbeitet bereits mit dem erhöhten `id`, während Transaktion Eins noch läuft. `open` ist eine Möglichkeit wie Transaktionen kommunizieren können.

Die Implementierung von `watch` ist recht trivial, aber `retry` ist interessant.

Listing 1.12. basic atomic block

```
// watch keyword adds an address to local wait set
void watch(Address a){
  VM_Thread.getCurrentThread().watchSet.add(a); }

// retry keyword implementation
void retry(){
  VM_Thread thread = VM_Thread.getCurrentThread();
  List watchSet = thread.watchSet;
  // register "cancel retry violation handler" to
  // cleanup scheduler if we violated before yield
  VM_Magic.registerViolationHandler(retryVH);
  for (int i=0,s=watchSet.size();i<s;i++){
      Address a=(Address)watchSet.get(i);
      open {
          // write address where scheduler is reading
          thread.schedulerAddress = a;
          // wakeup the scheduler violation handler
          thread.schedulerWatch = true; }
      // busy wait until we hear back
      open { if (thread.schedulerWatch) for(;;) ; }}
  // clear our read set to avoid violations
  // now that scheduler is listening for us
  VM_Magic.discardState();
  // safe to unregister now that read set cleared
  VM_Magic.unregisterViolationHandler(retryVH);
  // store resume context from checkpoint and yield
  thread.suspend(); }
```

Man beachte den zweiten `open` Block! Dieser Code wird innerhalb einer Transaktion ausgeführt. In ATOMOΣ gibt es einen Scheduler, der Transaktionen neu startet, falls sie invalidiert werden. In diesem Fall wird als die `open` Transaktion abgebrochen, sobald `thread.schedulerWatch` sich ändert. So kann der Code dann aus der „`for(;;) ;`“ Endlosschleife entkommen.

4.2 Optimierung

Ein interessanter Bereich im Übersetzer ist die Optimierung. Wir betrachten StarJIT [1], einen Java JIT Compiler, der versucht Transaktionen zu optimieren. Intern benutzt diese STM Implementierung die bereits erwähnte McRT-Bibliothek [9]. StarJIT wendet drei Optimierungsansätze an.

Existierende Optimierungen ermöglichen Der Javacompiler wendet verschiedene Methoden an, wie zum Beispiel das Entfernen von unerreichbarem Code. StarJIT fügt zusätzliche Prüfvariablen ein, damit die Semantik von `atomic` Blocks dabei nicht verletzt wird.

Globale Optimierungen Manche Variablen sind unveränderlich (zum Beispiel alle mit `final` gekennzeichneten). Diese müssen innerhalb von Transaktionen dann nicht auf Veränderungen überwacht werden.

Genauso können lokale Objekte von dieser Überwachung ausgenommen werden, wenn sie sich auf dem threadeigenen Stack befinden.

Codeerzeugung Zugriffe auf Transaktionen profitieren von optimiertem Assemblercode.

4.3 Funktionale Programmierung

Als abschließendes Kapitel wollen wir noch einen Blick auf Haskell werfen, eine rein funktionale Sprache. Für Ein- und Ausgabe verwendet Haskell das Konzept der Monade[7]. Für unsere Zwecke reicht Vereinfachung, dass man eine Monade als eine Art „Kontext“ ansieht.

Die STM-Monade funktionert wie die IO-Monade, man kann also die do-Notation verwenden und Sequenzen bilden. Die Schnittstelle zwischen den beiden „Kontexten“ bildet `atomic`.

Listing 1.13. Haskells atomic

```
atomic :: STM a -> IO a

# example
main = do { ...; atomic( readTVar x ); ...}
```

Das Typensystem von Haskell garantiert, dass `STM a` und `IO a` Aktionen sich nicht vermischen. Das Problem der Seiteneffekte ist also beseitigt.

Die Eigenschaften der Monade (do-Notation, Sequenzen) geben auch automatisch vor, wie man Transaktionen zusammenfügt.

Man stelle sich zwei Listen vor die intern durch eine Baumstruktur implementiert sind und synchronisiert wurden. Nun möchte man eine Funktion schreiben, die in einer Transaktion ein Element aus der einen Liste in die Andere verschiebt. Mit Sperren[8] ist das in der Form unmöglich.

Falls die Listen „transaktional“ implementiert sind, lässt sich die Anweisung so schreiben: `atomic(do  x <- delete t1 A; insert t2 A x)`

Man kann den Strichpunkt nicht nur als Terminator betrachten, sondern im Monadenkontext besser als Operator. Haskell bietet einen zusätzlichen Operator an, um Transaktionen zusammenzufügen. Normalerweise besteht eine Transaktion aus einer Sequenz von Anweisungen. Mit `orElse` kann man auch Verzweigungen einbauen.

Listing 1.14. orElse

```
atomic (readPort p1 `orElse` readPort p2)
```

Diese Transaktion versucht von einem Port `p1` zu lesen. Falls das fehlschlägt probiert sie vom anderen Port `p2` zu lesen. Falls beide fehlschlagen wird die Transaktion darauf warten, dass einer von beiden Ports verfügbar wird.

Transaktionaler Speicher ist also nicht nur eine Vereinfachung zur parallelen Programmierung sondern ein Konstrukt, das noch ganz andere Möglichkeiten bietet.

---

[7] Wikipedia: „Eine Monade ist im mathematischen Teilgebiet der Kategorientheorie eine Struktur, die gewisse formale Ähnlichkeit mit den Monoiden der Algebra aufweist“

[8] engl.: locks

5 Zusammenfassung

Wie in Kapitel 2 gezeigt wurde, sind noch einige Fragen offen. Trotzdem sieht es danach aus, das Transaktionaler Speicher ein gutes Mittel ist, um die kommenden Multicore Prozessoren auszunutzen. Die vielen Beispiele zeigen, dass bereits einige Ideen und Ansätze vorhanden sind, wie diese neue Methode für den durchschnittlichen Programmierer verwendbar gemacht werden kann.

Als Programmierer wäre es nun an der Zeit erste Versuche mit STM zu starten. Es muss in echten Programmen ausprobiert werden, um zu sehen, wie dieses neue Konstrukt sich bewährt.

Literatur

1. Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press.
2. Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press.
3. Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.
4. T. Harris. Exceptions and side-effects in atomic blocks, 2004.
5. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.
6. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.
7. Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM Press.
8. Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006. Condensed version submitted for publication.
9. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.