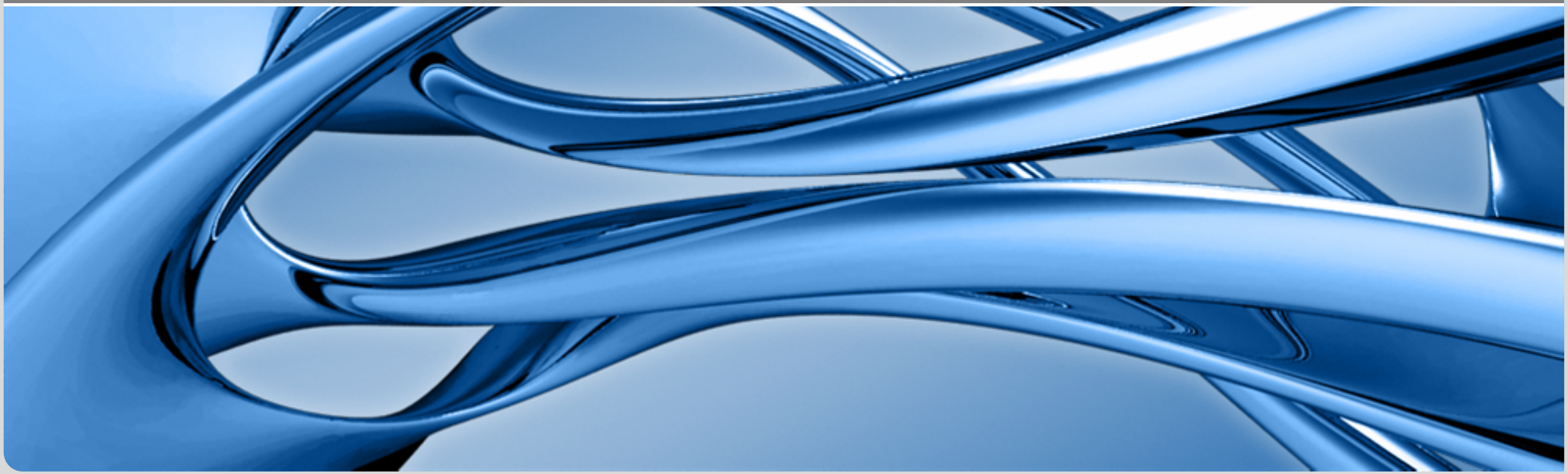
Karlsruher Institut für Technologie

Integration von Transactional Memory in Hochsprachen

Niklas Baumstark

Proseminar „Transactional Memory“

Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Prof. Karl



Gliederung

- Motivation
- C++
  - Designziele, Standardisierung
  - Semantik
  - Syntax
  - Optimierungen
- Haskell
  - Kurzer Überblick
  - Beispiel

Motivation

- TM deutlich einfachere Abstraktion als Locks:
  - `atomic { ... }` und fertig! (zumindest in der Theorie)
- Aber wie dem Programmierer zur Verfügung stellen?
  - Implementierung: Bibliothek vs. Compiler-/Runtimeintegration
  - Syntax: optional Spracherweiterung/Pragmas
  - Einfachkeit vs. viel Kontrolle (Performance!)
- Beispiele: C++, Haskell

TM in C++

- Designziele:
  - Geschwindigkeit
  - Flexibilität (C++ wird auf vielen Plattformen eingesetzt)
  - beliebige Speicherzugriffe in Transaktionen
  - C++-Sprachelemente unterstützen (Templates, Funktionszeiger, Methoden)
- SG5 beschäftigt sich mit der Integration von TM in C++1x [5]
- Basis: Spezifikations-Draft [1] (u.a. unterstützt von GCC 4.7 und Intel STM Compiler)
- Grundlegendes Design
  - Direkte Spracherweiterung für nahtlose Integration (keine Pragmas)
  - Compiler instrumentiert transaktionalen Code
    $\Rightarrow$ Load-/Store-Funktionsaufrufe an eine TM-Runtime
  - ABI der Runtime in [2] genau spezifiziert

TM in C++ (Semantik)

- Problematisch: Starke Atomarität
  - keine Zwischenergebnisse einer Transaktion sind für andere Threads sichtbar
    $\Rightarrow$ Also komplettes Programm instrumentieren?
- Einfacher: Single-Lock Atomicity (SLA) [7]
  - Programm verhält sich so, als ob ein globales Lock zur Serialisierung aller Transaktionen verwendet würde
  - I/O und Legacy-Code ist erlaubt
  - Probleme: Zurückrollen unmöglich, Isolationseigenschaft nicht garantiert

Geteilt: `atomic<int> x(0);`	
Thread 1	Thread 2

`transaction {`
`  x = 1;`
 	
`  while (x == 1) { }`
`}`
	
	`while (x == 0) { }`
	`x = 0;`

TM in C++ (Syntax)

- `__transaction_relaxed { ... }` mit SLA-Semantik
  - Kann beliebige Anweisungen enthalten
  - evtl. Wechsel in einen seriellen Ausführungsmodus (Funktionsaufruf an die Runtime)
- `__transaction_atomic { ... }` mit stärkerer Semantik
  - atomar im Kontext des Gesamtprogramms
  - können mit `__transaction_cancel` abgebrochen werden
  - keine irreversible Operationen (I/O, Aufrufe nicht instrumentierter Funktionen, Synchronisationsprimitiven)
- Transaktionen können geschachtelt werden (Ausnahme: `relaxed` nicht innerhalb von `atomic`)
- Ausdrücke und Funktionen können Transaktionen sein!

```
int x = 1;
int y = __transaction_atomic (x + 1);
```

TM in C++ (Codebeispiel)

```
void inc(int& x) { ++x; }
void plustwo(int& x) { inc(x); inc(x); }

int main() {
    int x = 0, y = 0;
    __transaction_atomic {
        plustwo(x);
        y = x; // x = y = 2
        __transaction_atomic {
            y += 2;
            if (y > 3)
                __transaction_cancel;
        }
    }
    __transaction_relaxed {
        std::cout << y << "\n";
    }
}

$ g++ -std=c++11 -fgnu-tm test.cpp && ./a.out
2
```

TM in C++ (Ungültige Codebeispiele)

```
int main() {
    atomic<int> i(0);
    __transaction_atomic {
        i = 1;
    }
}
```

```
$ g++ -std=c++11 -fgnu-tm test.cpp
test.cpp:2:14: error: unsafe function call
'std::__atomic_base<...>::operator=(...)' within atomic transaction
```

Geteilt: `int x = 0;`	
Thread 1	Thread 2

`__transaction_atomic {`
`  x = 1;`
`}`
	`x = 2;`

TM in C++ (Optimierungen)

- Compiler weiß durch statische Analyse viel über den Datenfluss. Naive Load-/Store-Funktionsaufrufe an TM-Runtime verhindern aber Optimierung
- Idee [6]: Nicht nur generische `read`/`write`-Operationen, sondern auch `read_after_read`, `write_after_write`, `read_after_write`, `read_for_write`
- Compiler teilt so sein Wissen der Runtime mit

Code	Repräsentation	Schritt 1	Schritt 2	Schritt 3

`y = x`
`x = x + y`
`return x`
	`read(x)`
	`write(y)`
	`read(x)`
	`read(y)`
	`write(x)`
	`read(x)`
		`read(x)`
		`write(y)`
		`readAR(x)`
		`readAW(y)`
		`writeAR(x)`
		`readAW(x)`
			`read(x)`
			`write(y)`
			`writeAR(x)`
				`readFW(x)`
				`write(y)`
				`writeAW(x)`

TM in der funktionalen Programmierung

- FP hat Vorteile in Bezug auf Parallelisierung und STM (z.B. Persistente Datenstrukturen)
- Aber: Programme kommen nicht komplett ohne veränderlichen Zustand aus
- Clojure, Haskell bieten erprobte STM-Bibliotheken
- Haskell
  - pur funktionale Programmiersprache, Seiteneffekte durch spezielle Typen (Monaden) repräsentiert
  - GHC (Glasgow Haskell Compiler): Green Threads und nichtblockierendes I/O $\Rightarrow$ eine Million Threads kein Problem
  - seit 2006: STM-Bibliothek in Kooperation mit der GHC-Runtime

Haskell's STM

- Wert vom Typ `STM a` ist eine Transaktion, die einen Wert vom Typ `a` produziert
- Transaktionen arbeiten auf Variablen vom Typ `TVar a`
- `retry` bricht Transaktion ab und führt sie erneut aus
- Klassisches Beispiel: Banktransaktion

```
transfer :: TVar Int -> TVar Int -> Int -> STM ()
transfer from to amount = do x1 <- readTVar from
                             x2 <- readTVar to
                             when (x1 < amount) retry
                             writeTVar from (x1 - amount)
                             writeTVar to (x2 + amount)
```

- Seiteneffektfreiheit der Transaktionen wird durch Typsystem forciert: I/O-Aktionen sind vom Typ `IO a`, es existiert keine Funktion

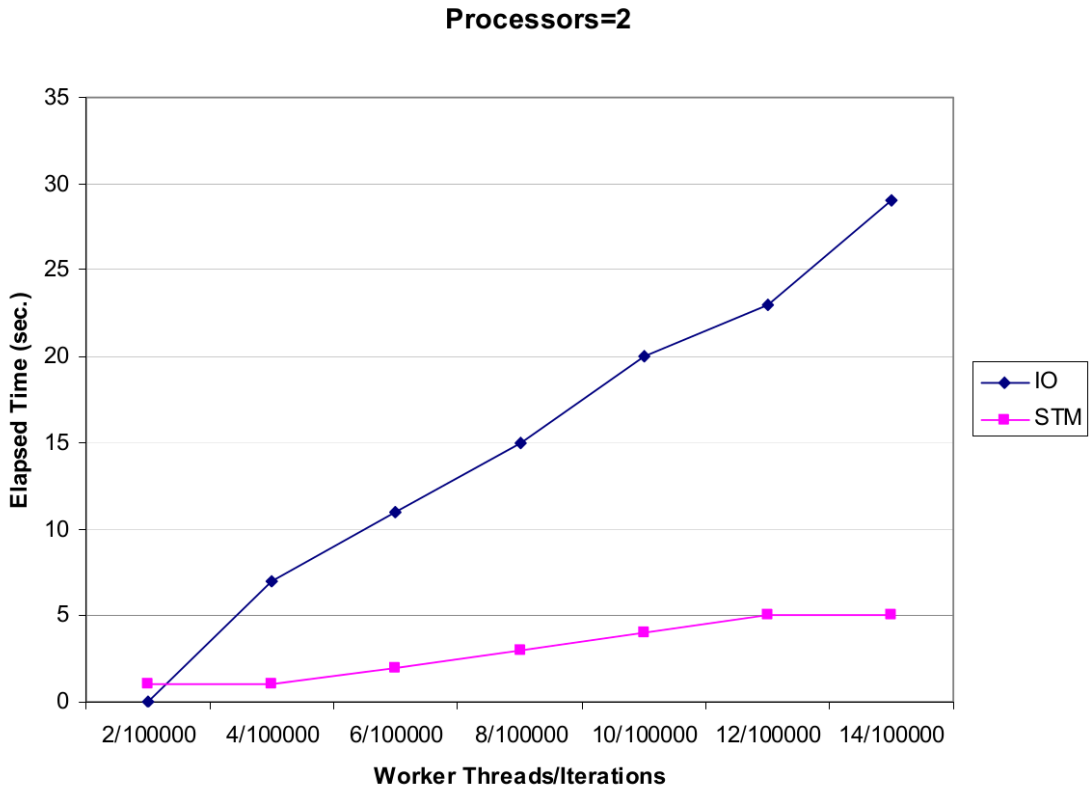
  `IO a -> STM a`
- Der umgekehrte Weg existiert: `atomically :: STM a -> IO a`

Haskell's STM

- Neben `TVar` z.B. Queues (`TChan`) und `TMVar`s (sind leer oder enthalten ein Datum)
- Kombinatoren wie `orElse :: STM a -> STM a -> STM a`

```
type Event = TMVar ()

waitFor :: Event -> STM ()
waitFor evt = takeTMVar evt

trigger :: Event -> STM ()
trigger evt = putTMVar evt ()

buttonClick :: Event
keyPress :: Event

waitForUserInput = waitFor buttonClick `orElse` waitFor keyPress
```

Haskell STM vs. Locking



- blockierenden Warteschlange: Locks (`IO`) vs. `STM` (Quelle: [4, Abbildung 2])
- Auch mit 8 Cores noch deutlich schneller [4]

Zusammenfassung

- Viele verschiedene Möglichkeiten, TM dem Benutzer zur Verfügung zu stellen
- Meiste Implementierungen erst in den letzten 7 Jahren entstanden, daher noch nicht sehr ausgereift, schnell und verbreitet
- In der Industrie noch eher wenig zu finden, hauptsächlich im Bereich HPC und in der funktionalen Programmierung
- Häufig maximale Performance nicht essentiell
  $\Rightarrow$ dann STM elegante und einfache Lösung!

Referenzen

[1] Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. Draft Specification of Transactional Language Constructs for C++, Version 1.1, February 2012.

[2] Intel® Corporation. Intel® Transactional Memory Compiler and Runtime Application Binary Interface, Revision 1.1 (Draft), May 2009.

[3] L. Crowl, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Integrating Transactional Memory into C++ In *The Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[4] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton-Jones, and Satnam Singh. Lock Free Data Structures using STMs in Haskell. In *FLOPS '06: Proceedings of the Eighth International Symposium on Functional and Logic Programming, to appear*, April 2006.

[5] Justin E. Gottschlich Michael Wong. SG5: Software Transactional Memory (TM) Status Report, August 2012.

[6] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and Implementation of Transactional Constructs for C/C++ *SIGPLAN Not.*, 43(10):195–212, October 2008.

[7] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards Transactional Memory Semantics for C++ In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 49–58, New York, NY, USA, 2009. ACM.

Karlsruher Institut für Technologie

Vielen Dank für die Aufmerksamkeit!

TM in C++ (Speedup durch Lese-Schreibzugriffs-Optimierung)

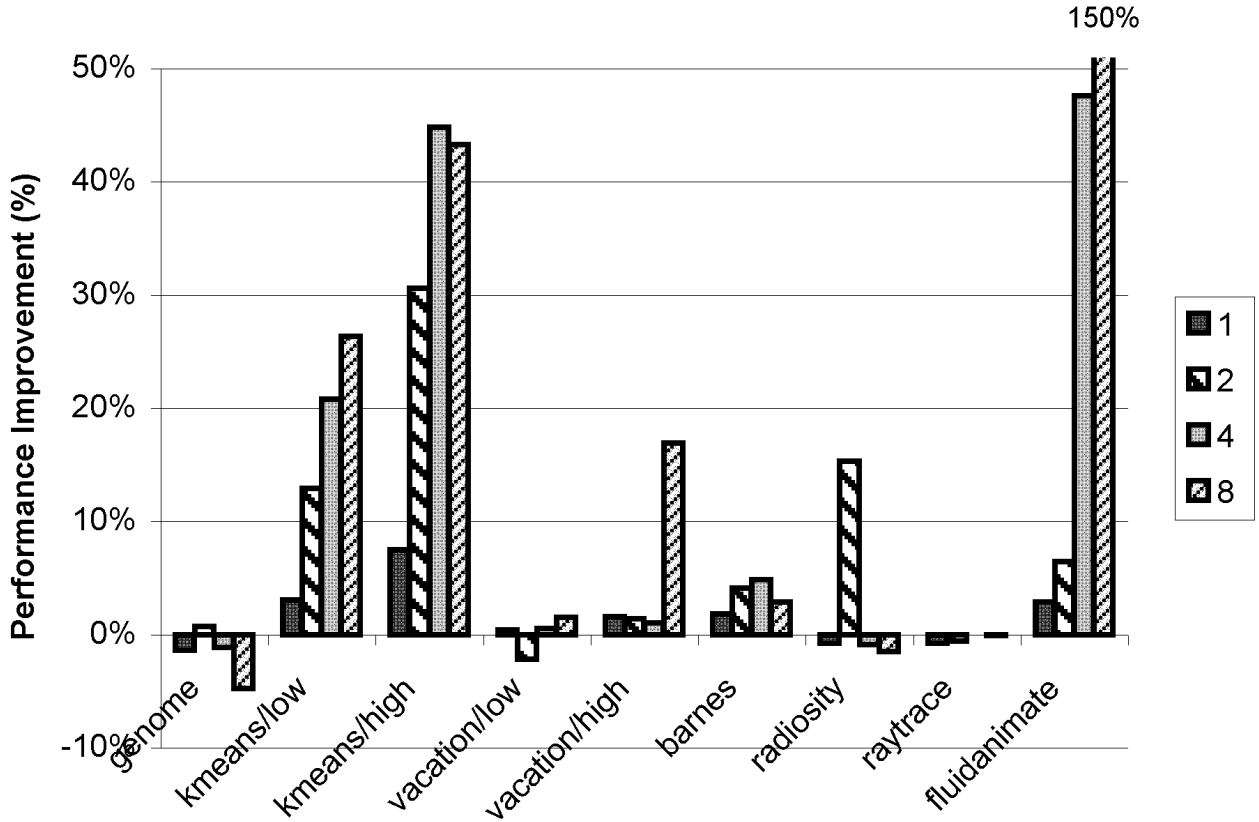


Abbildung : Quelle: [3, Abbildung 13]