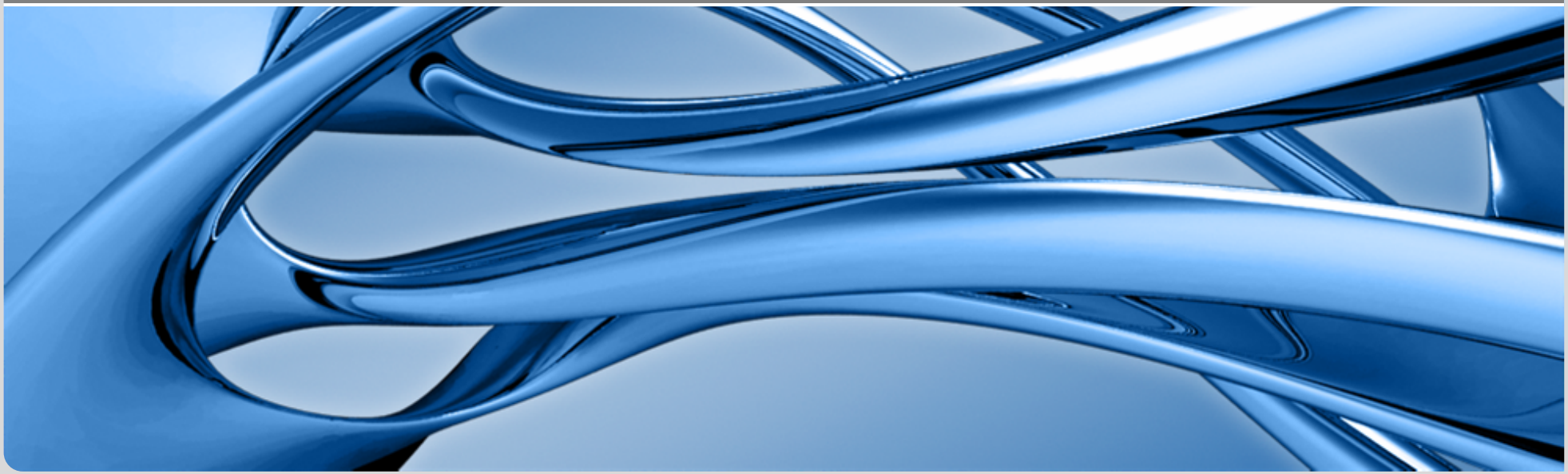
Karlsruher Institut für Technologie

Integration von Transaction Memory in Hochsprachen

Niklas Baumstark

Proseminar „Transactional Memory“

Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Prof. Karl



Gliederung

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

Einführung

- STM bereits für viele Programmiersprachen verfügbar
- Deutlich einfachere Abstraktion als Locks: `atomic { ... }`
- Verschiedene Designaspekte bei der Implementierung:
  - Implementierung: Bibliothek vs. Compiler-/Runtimeintegration
  - Syntax: optional Spracherweiterung/Pragmas
  - Performanceüberlegungen
- Beispiele: C++ und Haskell

TM in C++

- Designziele: Geschwindigkeit, Flexibilität, beliebige Speicherzugriffe in Transaktionen
- Seit Mai 2012: ISO WG21, SG5 beschäftigt sich mit der Integration von TM in den nächsten C++-Standard [6]
- Diskussionsbasis: Spezifikations-Draft [1], der u.a. von GCC 4.7 und Intel's STM Compiler bereits unterstützt wird
- Grundlegendes Design
  - Direkte Spracherweiterung für nahtlose Integration (keine Pragmas)
  - Compiler instrumentiert transaktionellen Code, indem er Lese-/Schreibzugriffe durch Load-/Store-Funktionsaufrufe an eine TM-Laufzeitumgebung ersetzt
  - Laufzeitumgebung ist austauschbar, ABI in [2] genau spezifiziert
- C++-Sprachelemente wie Templates, Funktionspointer, virtuelle Methoden etc. werden unterstützt

TM in C++ (Semantik)

- Starke Atomizität
  - keine Zwischenergebnisse einer Transaktion sind für andere Threads sichtbar
  - Transaktionen zurückrollbar
  - Nachteile:
    - kein I/O in Transaktionen möglich
    - Schwierig zu garantieren, da auch nicht transaktionaler Code involviert ist
- Einfacher: Single-Lock Atomicity (SLA) [5]
  - Programm verhält sich so, als ob ein globales Lock zur Synchronisation aller Transaktionen verwendet würde
  - I/O in Transaktionen möglich

TM in C++ (Syntax)

- `__transaction_atomic { ... }` mit starker Atomizität
  - laufen im Kontext des Gesamtprogramms atomar ab
  - keine irreversible Operationen (I/O, Aufrufe nicht instrumentierter Funktionen)
  - können mit `__transaction_cancel` zurückgerollt und abgebrochen werden
- `__transaction_relaxed { ... }` mit SLA-Semantik
  - Kann beliebige Anweisungen enthalten
  - Keine Isolation von nicht transaktionalem Code
  - Compiler muss evtl. Wechsel in einen seriellen Ausführungsmodus veranlassen (Funktionsaufruf an die Runtime)
- Transaktionen können ineinander geschachtelt werden (Ausnahme: `relaxed` kann nicht innerhalb von `atomic` stehen)

TM in C++ (Syntax)

- Instrumentierung ist teuer (Kompilierzeit, Codegröße) und nicht immer möglich (z.B. im Fall von I/O), daher zusätzliche Funktionsdeklarationen
  - `transaction_callable`: Empfiehlt dem Compiler, eine instrumentierte Variante der Funktion zu erzeugen, die dann in `relaxed`-Transaktionen verwendet wird. Optional.
  - `transaction_unsafe`: Funktion kann nicht in `atomic`-Transaktion verwendet werden. Implizit angenommen.
  - `transaction_safe`: Funktion kann in `atomic`-Transaktion verwendet werden. Funktion muss gewisse notwendige Bedingungen erfüllen. Kann in gewissen Fällen auch inferiert werden!
- auch für Lambdas, Konstruktoren, Funktionszeiger, Initialisiererausdrücke etc.
- können für verschiedene Überladungen oder Templateinstanzen unterschiedlich sein

TM in C++ (Codebeispiel)

```
__attribute__((transaction_safe)) int inc(int &x) { return ++x; }
int f(int& x) { return 2*inc(x); } // implizit transaction_safe

int main() {
    int x = 0, y = 0;
    __transaction_atomic {
        y += f(x);  // x = 1, y = 2
        __transaction_atomic {
            y += 2;
            if (y > 3)
                __transaction_cancel;
        }
    }
    __transaction_relaxed {
        std::cout << y << "\n"; // => 2
    }
}
```

TM in C++ (Optimierungen)

- Compiler weiß durch statische Analyse viel über den Datenfluss. Naive Load-/Store-Funktionsaufrufe an TM-Runtime verhindern aber Optimierung
- In [7] wird bereits sehr ähnliche Implementierung diskutiert, ebenfalls mit einer externen TM-Runtime
- Idee dort: 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 C++ (Speedup durch Lese-Schreibzugriffs-Optimierung)

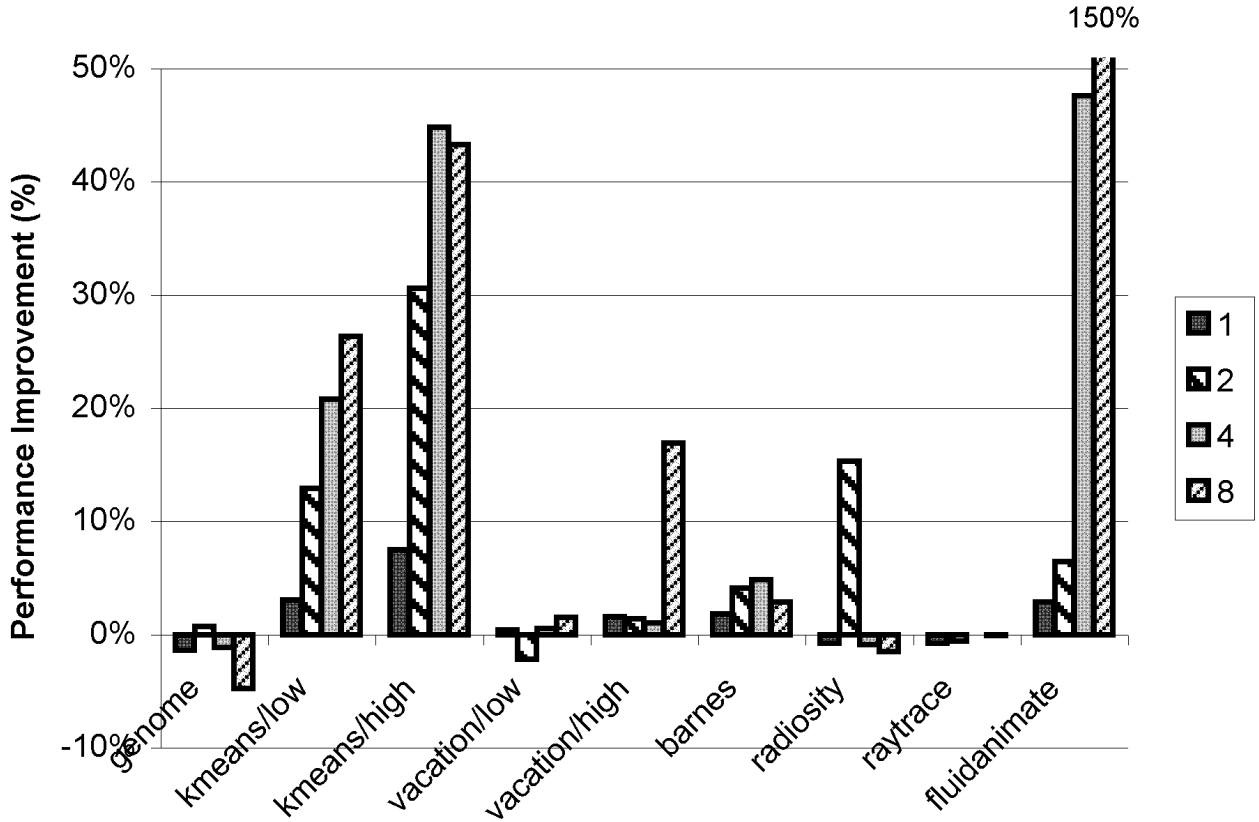


Abbildung : Quelle: [3, Abbildung 13]

TM in der funktionalen Programmierung

- Funktionale Konzepte haben Vorteile in Bezug auf Parallelisierung (Zugriff auf persistente Datenstrukturen muss nicht koordiniert werden)
- Aber: größere Programme kommen nicht komplett ohne veränderlichen Zustand aus
- Clojure, Haskell, inzwischen auch Scala bieten ausgereifte STM-Bibliotheken
- Haskell
  - pur funktionale Programmiersprache, Seiteneffekte (und auch STM) werden mit Monaden modelliert
  - Green Threads und nichtblockierendes I/O erlauben hohe Nebenläufigkeit in Programmen (eine Million Threads kein Problem auf einem normalen Laptop)
  - Glasgow Haskell Compiler (GHC) liefert seit 2006 eine STM-Bibliothek mit (die in Kooperation mit der GHC-Runtime funktioniert)

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, wenn sich die Werte der gelesenen Variablen ändern
- 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` und es existiert keine Funktion `IO a -> STM a`
- Der umgekehrte Weg existiert: `atomically :: STM a -> IO a`

Haskell's STM

- Neben `TVar` existieren noch Container wie Queues (`TChan`) und `TMVar`s (können leer sein oder ein Datum enthalten)
- Mächtige Kombinatoren wie `orElse :: STM a -> STM a -> STM a` erlauben sehr kompakte Lösungen für Probleme, die mit klassischem Locking nur umständlich zu lösen sind

```
parallelSearch :: [a] -> (a -> Bool) -> IO a
parallelSearch lst pred =
    do chan <- newTChanIO
       result <- newEmptyTMVarIO
       forkIO (worker chan result)
       forkIO (worker chan result)
       mapM (\x -> atomically (writeTChan chan x)) lst
       atomically (readTMVar result)
    where worker chan res = do x <- readTChan chan
                               when (pred x) (putTMVar res)
```

Haskell's STM

```
-- Events mit STM
type Event = TMVar ()

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

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

onButtonClicked :: Event
onKeyPressed :: Event
onUserInput = onButtonClicked `orElse` onKeyPressed
```

Haskell STM vs. Locking

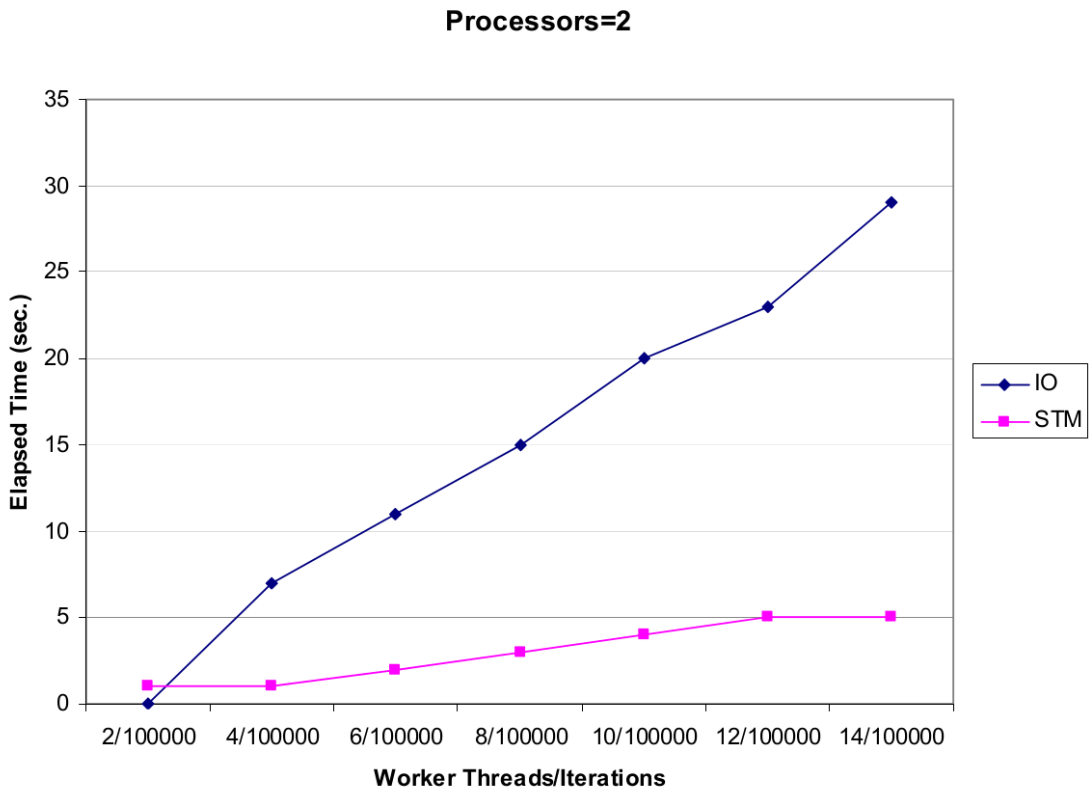


Abbildung : Performance von Haskell STM im Vergleich zu traditionellem Locking am Beispiel einer blockierenden Warteschlange (Quelle: [4, Abbildung 2])

Zusammenfassung

- Viele verschiedene Möglichkeiten, TM dem Benutzer zur Verfügung zu stellen
- Die meisten 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 ist maximale Performance nicht essentiell, dann ist STM eine 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] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. *SIGPLAN Not.*, 43(5):15–26, May 2008.

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

[7] 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.