# Memoria transazionale


alla ricerca del Graal
della programmazione concorrente

Massimiliano Ghilardi

# Programmazione concorrente


## Perché?

- legge di Moore (1965) ancora valida
- le prestazioni dei singoli core non aumentano più circa dal 2005

## Processori desktop sempre più multi-core

- Dual-core — AMD Athlon 64 X2 (2005), Intel Pentium D (2005)
- Quad-core — AMD Opteron 8xxx (2007), Intel Xeon X 32xx (2007)
- Octa-core — AMD Opteron *Magny-Cours* (2010), Intel Xeon E7xxx (2008)
- 12-core — AMD Opteron *Magny-Cours* (2010), Intel Xeon E5-269x v2 (2013)
- 16-core — AMD Opteron *Interlagos* (2011)

# Programmazione concorrente


Come?

Programmi multi-thread

- unità di esecuzione distinte, che condividono la stessa memoria
- passaggio dati da un thread ad un altro semplice e veloce:
  basta leggere e scrivere in memoria
- strutture dati thread-safe (es. Java ConcurrentHashMap)
- locking (mutua esclusione) per coordinare manualmente l'accesso
- tecniche avanzate: operazioni atomiche, barriere di memoria

# Programmazione concorrente

Sembra andare tutto bene...

# Problemi concorrenti


”Concurrent threads are notoriously hard to use
– and debug – due to their inherent nondeterminism”

Problems:

- Deadlock
- Livelock
- Starvation
- Priority inversion
- Race conditions
- Nondeterminism
- Non-composability

Showing an algorithm correctness is exponentially difficult => Bugs

# Problemi concorrenti


”I thread concorrenti sono notoriamente difficili da usare – e  debuggare – per il loro intrinseco non-determinismo”

Problemi:

- Deadlock — stallo
- Livelock — ”Prima tu.” ”No, prima tu.”
- Starvation — inedia
- Priority inversion — inversione di priorità
- Race conditions — corse critiche
- Nondeterminism — non-determinismo
- Non-composability — non-componibilità

Difficoltà esponenziale nel mostrare la correttezza di un algoritmo => Bachi

# Problemi concorrenti

Forse non va così bene...

# Catastrofi concorrenti?


- Java Concurrency in Practice [2006]
  oltre 400 pagine: quasi tutte sulle corse critiche,
  come riconoscerle e come evitarle
- la progettazione del codice multi-threaded è esponenzialmente difficile
- il debug interattivo pure
- anche l'analisi e la risoluzione dei bug
- i bug non-deterministici sono la norma
- i test non sono mai esaustivi

Altro?

- le API thread-safe basate sui lock non sono componibili

# Esempi concorrenti


I filosofi a cena (dining philosophers, Dijkstra 1965)

- N filosofi, N piatti ed N bacchette attorno ad un tavolo rotondo
- Ogni filosofo può mangiare solo se ha preso entrambe le bacchette ai lati del suo piatto
- Ogni bacchetta è condivisa tra due filosofi

L'implementazione ovvia con i lock nasconde delle sorprese

# Esempi concorrenti

I filosofi a cena (dining philosophers, Dijkstra 1965)

Volontari?

# Modelli alternativi


## Il problema principale è la memoria condivisa

- Semplice e veloce, nasconde insidie esponenziali
- L'illusione di causalità è a rischio – e non siamo fatti per ragionare senza di essa
- Rischio continuo di corse critiche = non-determinismo

## Soluzioni alternative?

- π-calculus
- message passing
- futures, API asincrone o ad eventi
- memoria transazionale
  http://en.wikipedia.org/wiki/Software_transactional_memory

# Memoria transazionale


I filosofi a cena (dining philosophers, Dijkstra 1965)

## Versione transazionale

# Memoria transazionale


## Concetti fondamentali

- Ogni thread può eseguire blocchi `atomic { … }` (A di ACID)
- Ogni transazione può essere abortita (rollback) volontariamente o se non è possibile garantire la consistenza (C di ACID)
- Ogni transazione è completamente isolata dalle altre (I di ACID)
- Di solito usa internamente i lock (o meglio compare-and-swap) per implementare il commit
- Di solito usa internamente le barriere di memoria

## Funzionalità avanzate

- `retry()`
- `orelse { … }`
- `nonblocking { … }`

# Memoria transazionale


Come trasformare codice single-thread
in codice transazionale

- Aggiungere blocchi `atomic { … }`
- Dove necessario, aggiungere `@transactional` alle proprie classi
- Dove necessario, sostituire le normali strutture dati con le equivalenti strutture transazionali

Sembra troppo facile?

# Svantaggi


- Il codice nei blocchi `atomic { … }` può essere eseguito più volte del previsto, es. in caso di conflitti.

- Perciò operazioni irreversibili come
  - input/output
  - lanciare un missile (!)

  sono una pessima idea. Soluzione: accodare tali operazioni in un buffer transazionale ed eseguirle successivamente.

- Le transazioni software sono relativamente lente
  - overhead tipico 50-100 nanosecondi
  - le singole letture e scritture sono 5-10 volte più lente

  Problema risolto dalle transazioni hardware, es. **Intel TSX**

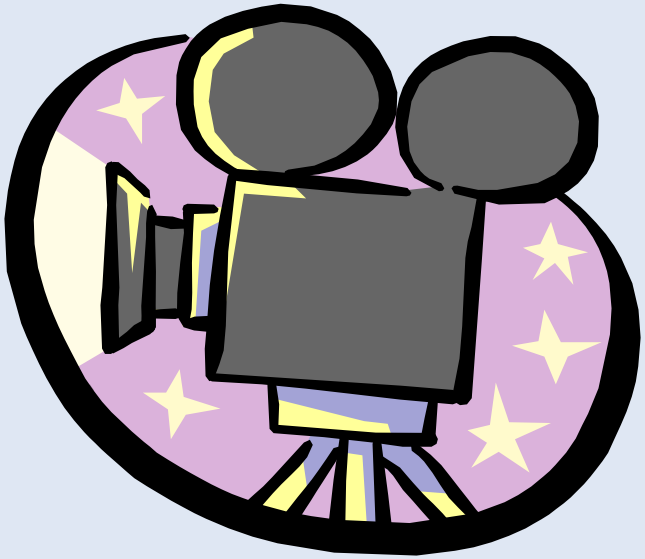
# Implementazioni


- C/C++ ~10 librerie. compilatori **Gnu GCC** e **Intel**
- C# 5-6 librerie
- Clojure built-in
- Common Lisp 2 librerie, tra cui **STMX**
- F# 1 libreria
- Groovy 1 libreria
- Haskell 2 librerie
- Java 5-6 librerie, tra cui **Deuce**
- Javascript 1 libreria
- OCaml 1 libreria
- Perl 1 libreria
- Python 3 librerie
- Scala 4 librerie

# STMX

- scritta dal relatore
- una delle prime, se non la prima, che supporta le transazioni hardware

# Dimostrazione dal vivo



# Domande?

Grazie per l'attenzione!

STMX http://stmx.org/

e-mail massimiliano.ghilardi at gmail.com

# Un po' di storia


fine anni '70: definizione del concetto di transazione per i DBMS [Jim Gray]

1983: Acronimo ACID per le proprietà delle transazioni (Atomicity, Consistency, Isolation, Durability) [Andreas Reuter, Theo Härder]

1986: Idea: transazioni in Lisp come paradigma di programmazione concorrente, realizzando apposite istruzioni della CPU (transazioni hardware). Articolo "An architecture for mostly functional languages" [Tom Knight]

1993: L'idea delle transazioni hardware si diffonde. Articolo "Transactional memory: architectural support for lock-free data structures" [Maurice Herlihy and J. Eliot B. Moss]

1997: Altra idea: transazioni puramente software. Il supporto hardware è ancora un miraggio... Articolo "Software Transactional Memory" [Nir Shavit, Dan Touitou].

2002-2006: Decine di articoli sull'argomento. Microsoft e Sun (adesso Oracle) creano gruppi di ricerca. Prime implementazioni pratiche.

2005: Articolo "Composable Memory Transactions" [Microsoft Research] è pieno di codice Haskell e pseudo-codice, spiega in dettaglio come implementare le transazioni software. Rilasciata implementazione Haskell.

2006: Articolo "Transactional Locking II" [Dave Dice, Ori Shalev, Nir Shavit] finalmente risolve un problema fondamentale: la consistenza ("C" di ACID).

2007-2012: Continua la pubblicazione di articoli, spesso affiancati da implementazioni. Riaffiora l'idea del supporto hardware.

Febbraio 2012: Intel annuncia il supporto alle transazioni hardware per l'anno successivo, le "Transactional Synchronization Extensions" (TSX).

Marzo 2013: Articolo "Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory" [Alexander Matveev, Nir Shavit] è pieno zeppo di pseudo-codice, spiega in dettaglio come rendere compatibili transazioni hardware e software, realizzando un sistema ibrido.

Marzo 2013: iniziata STMX, libreria Lisp di transazioni software.

Giugno 2013: Intel immette sul mercato i primi Core i5 e Core i7 con istruzioni TSX.

Luglio 2013: aggiunto a STMX il supporto per istruzioni TSX.

Agosto 2013: aggiunto a STMX il supporto per transazioni ibride.