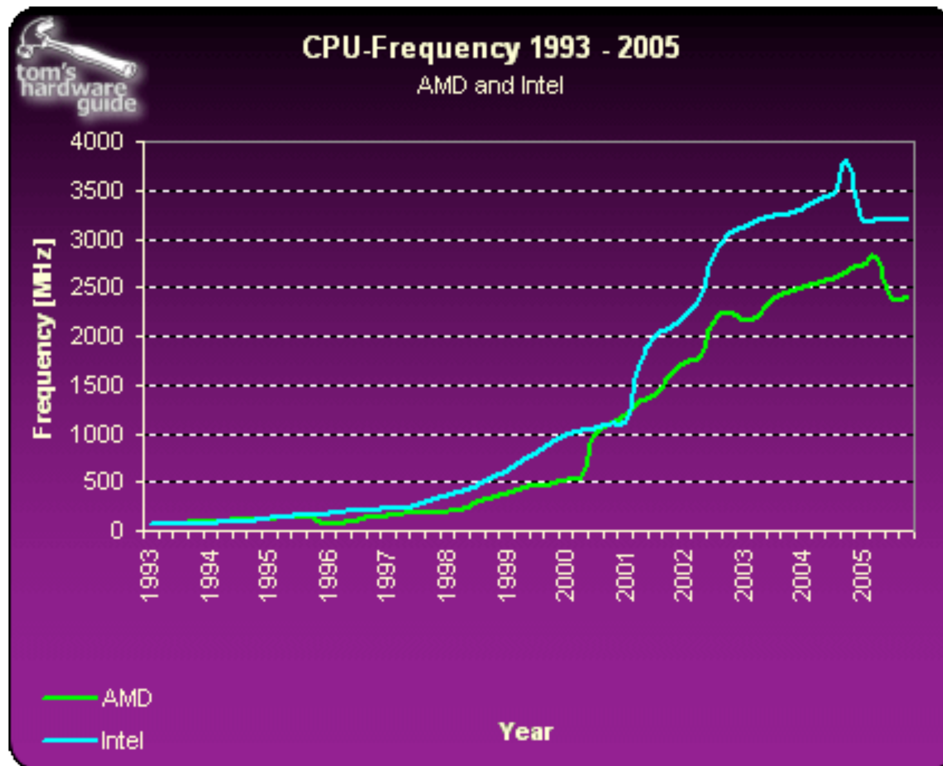
# Potential Show-Stoppers for Transactional Synchronization

Panel session, PPoPP'07, March 2007

| | |
|---|---|
| Michael L. Scott | U Rochester |
| Ali-Reza Adl-Tabatabai | Intel Corp |
| David Dice | Sun Microsystems |
| Christos Kozyrakis | Stanford U |
| Christoph von Praun | IBM Research |

# Uniprocessor Limits

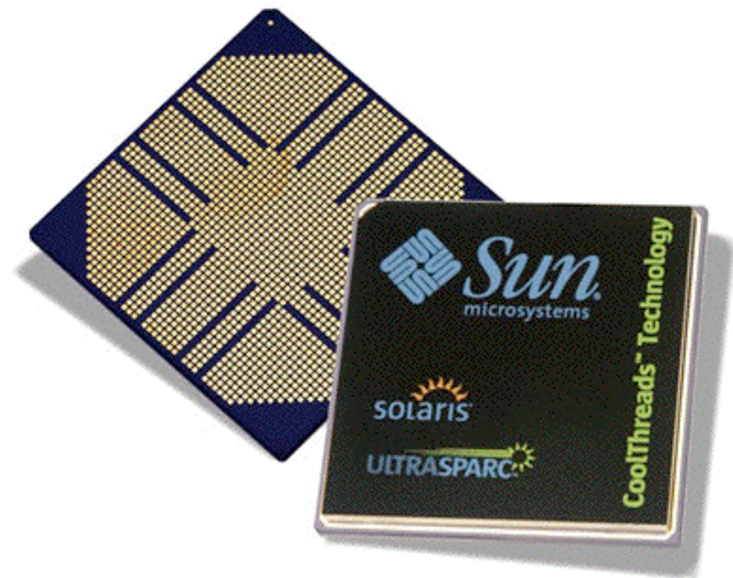



- Heat wall
- Limited ILP

http://www.tomshardware.com/2005/11/21

# Multicore is here to stay

- Dual-processor laptops now
- Quad-core desktops
- 8-core servers
- Lots more to come
- Vendors waiting for apps



# The Coming Crisis

- Parallelism common in high-end scientific computing
  - done by experts, at great expense
- Also common in Internet servers
  - "embarrassingly parallel"
- Has to migrate into the mainstream
  - programmers not up to the task

http://tfp.killbots.com/?p=wall/@wall&name=&pag=3

# What TM is

- A way to simplify some forms of synchronization — an alternative to mutual exclusion locks
- A way to improve scalability with respect to *coarse-grain* locks

# What TM is *not*

- A way to make parallel programming easy
- A general-purpose synchronization mechanism
- A way to get free concurrency (or even scalability)

# The basic idea is simple

- Programmer identifies atomic sections
- System serializes them, runs in parallel if it can

# Some details are *not* simple

- I/O and other irreversible operations
- Open nesting: causality loops, compensating actions, high-level concurrency control
- Weak isolation, privatization
- Early release
- Condition synchronization (retry, ...)
- Alternative paths (or else, ...)
- Customizable backoff or retry policies
- Synchronizers or other cross-transaction communication
- Priorities
- Segregation of transactional and nontransactional objects or types, for the benefit of SW implementations

# Not to mention

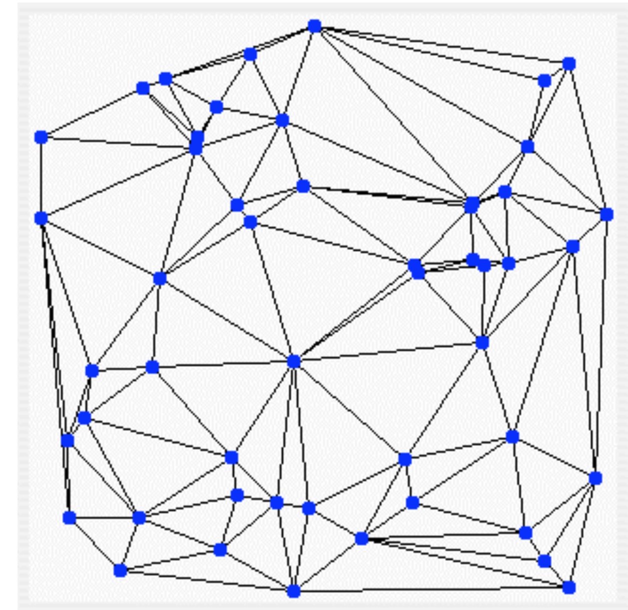
- Parallelization / identification of speculative tasks
- Ordering among transactions
- Performance tuning
  - » tools to find conflicts
  - » incentive to subdivide to avoid them

- When does this get uglier than locks? (answer: very quickly)

→ danger of overselling

# Some personal experience

- Delaunay mesh application
  - 2500 lines of C++
  - barrier-separated private and transactional phases
- RSTM library-based STM
  - transactional types inherit from transactional base class
  - access through smart pointers



- Turned out to be a lot harder than I expected

# A compiler would have helped

- Hide accessors, validators
- Generate transactional and non-transactional versions of code as needed
- Let `this` be a smart pointer
- Leave immutable fields in place, for safe private access; update read-only pointers as needed; support safe break/return
- Catch loop-carried private value, potentially stale private pointer
- Elide redundant checks

★ All of this is straightforward

# The Bottom Line

- Keep it simple!
- Don't expect too much
- Plan on language integration and compiler support

- Do not oversell !

# TRANSACT'07

## The Second ACM SIGPLAN Workshop on Transactional Computing

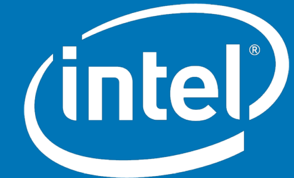
To be held in conjunction with **PODC 2007**

Portland, Oregon, August 16, 2007

**Submission deadline: April 15, 2007**

www.cs.rochester.edu/meetings/TRANSACT07/

This page intentionally left blank.



# Potential Show-Stoppers for Transactional Synchronization

## PPoPP '07 Panel Session

Ali-Reza Adl-Tabatabai

Programming Systems Lab

Intel Corporation

# Killing the Feng Shui

TM promised to bring harmony

- Programmer declares atomicity
- System implements under the hood

But we made compromises

- Lock-free → lock-based
- Isolation & memory ordering
- Explicit locking & compensating actions
- Explicit function annotations
- Virtualized HW TM → HW acceleration

And we've only just begun...



intel®

# More challenges remain

- Language & library integration
- Handling I/O
- Nested parallelism
- Communication
- Handling legacy code
- Real applications & large transactions
- Contention management
- Performance predictability
- Single thread overheads
- Performance & debug tools
- External transaction managers
- . . . I probably forgot something



intel®

Will transactions provide enough value when we're done?



intel

# The brighter side

- Databases have used transactions successfully for years
  - There’s more we can learn here
- New languages supporting transactions from ground up
  - Fixes some of the warts
- TM HW has other uses
  - Speculative threading
  - Speculative optimizations
  - Speculative lock elision
- STMs might enable new features
  - Debugging



intel®

# Parting thoughts

- We’ve compromised some of TM’s elegance
- More research challenges remain
- Will it provide enough value over locks when we’re done?
- Under promise so we can (over) deliver



intel®

This page intentionally left blank.

# Show-stoppers for Transactional Memory

***Dave Dice – blogs.sun.com/dave***

***J2SE Core Engineering***

***SunLabs Scalable Synchronization Group***

PPoPP Panel 2007-3-15

# Concurrency

- Here today
- Explicit thread-level parallelism
  - **not** a future
  - a remedy with side-effects
  - brings hope of performance
  - and promise of complexity
  - end of the lay-z-boy programming era (David Patterson)

# Human scalability

- Today:
  - lots of available cores
  - small concurrency priesthood
- Programs – programmers
- Reduce **complexity**
  - Eliminate common sources of errors
  - Think sequentially, execute concurrently
  - At least raise the abstraction level above locks

# TM Critique

- Restrictions (as of today)
  - large/long transactions
  - IO and irrevocable state
- Single-threaded latency ?
  - yes, it's important
- Missing infrastructure:
  - debugging, performance profiling
- Open issues:
  - atomicity, nesting, exceptions

# Better than locks ?

- Wish: synchronized ~~(Lock)~~ {...}
- **Not** a drop-in-replacement
- decreased complexity; added constraints
- Better but not good enough
- Transactions won't displace locks
  - incremental adoption
- We'll end up with both
  - lock-aware transactions?

# A useful addition?

# Shared Mutable State

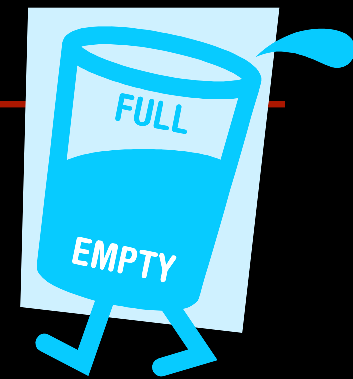
- Minimize shared mutable state
- Locks and transactions : immutable view
- Eliminate shared data
- **Message passing**: MPI, Erlang, etc
- 1 thread per address space
- Same programming model inter- & intra-node
- Can't express common concurrency bugs
- Can you express large systems?
  - old-school distributed programming

# Where does this take us?

- Locks + transactions + message passing
- Keep the lock abstraction
  - Transparently **Commute** to transactions
  - Revert to actual locks only as needed
  - Complexity of coarse-grained locking
  - Possibly better performance

This page intentionally left blank.

# ~~Potential Show-Stoppers for Transactional Synchronization~~




**Christos Kozyrakis**

Computer Systems Lab
Stanford University

http://csl.stanford.edu/~christos

# Ok, the base TM ideas look good; what's next?

**Christos Kozyrakis**

Computer Systems Lab
Stanford University

http://csl.stanford.edu/~christos

# 1. Apps & User Studies

- Are we really simplifying parallel programming?
  - Let’s write new apps or get feedback from others
- What are the common cases and pattern?
  - This is what we’ll make simpler, faster, …
  - Are we sure TM is sufficient to address all of them?
- Casting lock-based apps in TM is <u>dangerous</u>
  - Will fine-grain, rare transactions be common?

# 2. atomic{} is a primitive, not a parallel programming model

- DB users program SQL, not atomic{}
- Need truly high-level programming models
  - Simple & declarative like SQL, Mapreduce, …
  - atomic{} will be critical in implementing them
  - But it will probably take more than atomic{}
    - Primitives for finding concurrency and handling locality, coordination, scheduling, balance…
- Prog. environment = language + tools + libs
  - Use TM to build better debugging/tuning tools
  - See talk in next session for the libs issue

# 3. Atomicity ≠ Coordination

- TM is not a hammer for every nail
- Lots of work on forcing coordination into TM
  - Open-nesting, escape actions, non-isolated transactions, dependent transactions, …
  - Use semantics get really ugly, really quickly
  - Is it worth it? What do we expose to user and how?
- Simpler idea: use TM for what it is
  - Transactions = atomicity + isolation
  - Combine with other primitives to address other problems

# 4. Transactional memory & I/O

- TM is not a hammer for every nail
  - We can have restricted I/O within TM but…
- Better idea: make TM work with other transaction resources in the system
  - DB, LFS, message queues, …
  - System-level manager coordinates user transaction across all resources
  - Easier-to-use, flexible model with some restrictions
- Can this ever work?
  - Look at IBM's Quicksilver project

# 5. Beyond concurrency control

- Atomicity & isolation are generally useful
  - For debugging, profiling, checkpointing, exception handling, garbage collection, security, speculation …
- These may be TM's initial "killer apps"
- But they also change the requirements
  - Cheap transactions for pervasive use
  - "All transactions, all the time"

# Miscellaneous TM Issues

- Language support: YES
- Compiler support: YES
- HW support: YES
- Strong atomicity: YES
- Contention management: YES
- Compensating actions: YES
- High-level concurrency control: YES
- …

- ***9am*** **panels: NO**

This page intentionally left blank.

IBM T.J. Watson Research Center

# Potential Show-Stoppers for Transactional Synchronization

Christoph von Praun

PPoPP – Panel, March 15, 2007

1) Technical Challenges for TM

2) Environment, “Killer Apps“

# Technical challenges for TM

- Semantics and simplicity of the programming interface:
  - handling of irreversible operations, compensation actions
  - modularity and nesting
  - conditional synchronization, communication with concurrent transactions
  - interaction of transactional and non-transactional code
  - large transactions, contention management

- Performance and implementation:
  - reduce overheads
  - ‘right’ combination of software and hardware mechanisms

→ tremendous progress over the past years

# Multicore workloads (1/2)

## Web-Services

- *The* growth field in commercial computing:
  - large investments that can drive technological advances
  - lots of web-service developers from emerging economies
- Programming model:
  - “containerized” application frameworks, e.g., J2EE (concurrency not exposed to programmer)
  - “shared nothing architectures”, e.g., PHP, Ruby on Rails, ...

→ very high pressure to develop scalable middleware

# Web-Services continued ...

- Middleware is tuned for scalable concurrency *now.*
- Alternative technologies to enable scalable concurrency are becoming common practice:
  - non-blocking algorithms, libraries for concurrency utilities
  - advanced locking schemes
  - speculative lock elision
  - read-copy-update, ...
- The bar for TM is rising: TM has to offer *very significant advantage* over alternative technologies to justify cost of change.
  - better programmability
  - higher performance

→ IT moves fast, timing matters

→ TM currently behind the train

# Multicore workloads (2/2)

## Scientific applications

- Focused usage context
  - programmers willing to rewrite some code
  - semantic limitations of TM are acceptable
- Users care about *performance*
- Parallel computing and algorithms are established in the community
  - several factors can limit scalability, TM may solve one of them

## Game workloads [Tim Sweeney, POPL'06]

- Focused usage context
  - (S)TM seems right match for parallel game simulation
  - alternatives to transactional synchronization are unattractive
- Users care about *simplicity of the programming interface, programmability* (rapid development)

## Summary

- TM is a great technology
  - technical challenges are not show-stoppers
- Success or failure of TM not only decided on technical merit
- Critical for widespread adoption of novel technology (TM) is economic context (need “killer-application”)
- Different domains have different challenges:
  - middleware for web-services: *timing*
  - scientific applications: *performance*
  - games: *simplicity of the programming interface, programmability*

praun@us.ibm.com