Transactional Language Constructs for C++, and C, and Fortran (maybe)

vs.

WG21 SG5 C++ Standard TM Sub-Group

Michael Wong
michaelw@ca.ibm.com

International Standard Trouble Maker, Chief Cat Herder
OpenMP CEO
Canada and IBM C++ Standard HoD
Vice chair of Standards Council of Canada Programming Languages
Chair of WG21 SG5 Transactional Memory
Director and Vice President of ISOCPP.org

Acknowledgement and Disclaimer

- Numerous people internal and external to the original C++ TM Drafting Group, WG21 SG5 TM group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.
- I even lifted this acknowledgement and disclaimer from some of them.
- But I claim all credit for errors, and stupid mistakes. **These are mine, all mine!**
- Any opinions expressed in this presentation are my opinions and do not necessarily reflect the opinions of IBM.

Agenda

- **STM, HTM, HybridTM**
- Birth of a specification
- Design Goals
- Motivation for SG5 in C++ Standard
  - Use cases
  - Usability
  - Performance
- Language Constructs
  - Transactions, atomic and relaxed
  - Race-free semantics
  - Unsafe statements
  - Attributes
  - Transaction expressions and try blocks
  - Cancel
  - Exception handling
- SG5 Progress

Where were you in 1993?

- John Major was Prime Minister of Great Britain
- Brian Mulroney, Kim Campbell, Jean Chrétien were Prime Ministers of Canada
- Bill Clinton was the President Of U.S.
- EU was formally established by the Treaty of Maastricht, Helmut Kohl and Francois Mitterand
- Intel Pentium chip was the hot chip
- World Wide Web Mosaic browser was the hottest software around
- Maurice Herlihy and Elliot Moss wrote
  - Transactional Memory: Architectural support for lock free data structures
  - And then got < 10 citations/yr UNTIL 2005: not impressive
  - 2005: Multicore is coming: there is no more free-lunch!
  - Now you get 80000 hits on google, 2.7 mil on Bing

Where is transactions in the grand scheme of Concurrency

	Asynchronus Agents	Concurrent collections	Mutable shared state
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory
examples	GUI,background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)
key metrics	responsiveness	throughput, many core scalability	race free, lock free
requirement	isolation, messages	low overhead	composability
today's abstractions	thread,messages	thread pools, openmp	locks, lock hierarchies
future abstractions	futures, active objects	chores, parallel STL, PLINQ	transactional memory, declarative support for locks

Transactional Memory

- Transactions appear to execute atomically
- A transactional memory implementation may allow transactions to run concurrently but the results must be equivalent to some sequential execution
- Just a form of optimistic speculation

Example

```
Initially, a == 1, b == 2, c == 3
```

T1

```
atomic {
  a = 2;
  b = a + 1;
  c = b + 1;
}
```

T2

```
atomic {
  r1 = a;
  r2 = b;
  r3 = c;
}
```

```
T2 then T1 r1==1, r2==2, r3==3
```

```
T1 then T2 r1==2, r2==3, r3==4
```

T1

```
a = 2;
b = 3;

c = 4;
```

T2

```
r1 = 1

r2 = 3;
r3 = 3
```

```
Incorrect r1==1, r2==3, r3==3
```

Lock elision

```
synchronized  {
 node.next = succ;
 node. prev = pred;
 node. pred = node;
 node.next = node;
}
```

Why TM?

- A transaction is an atomic sequence of steps
- Intended to replace locks and conditions
- A better way to build lock-free data structures
  - CAS, LL/SC only works on a memory location, or at best a contiguous memory atomically
  - But there is no way to atomically alter a set of non-contiguous memory locations

What is Transactional Memory (Again) ?

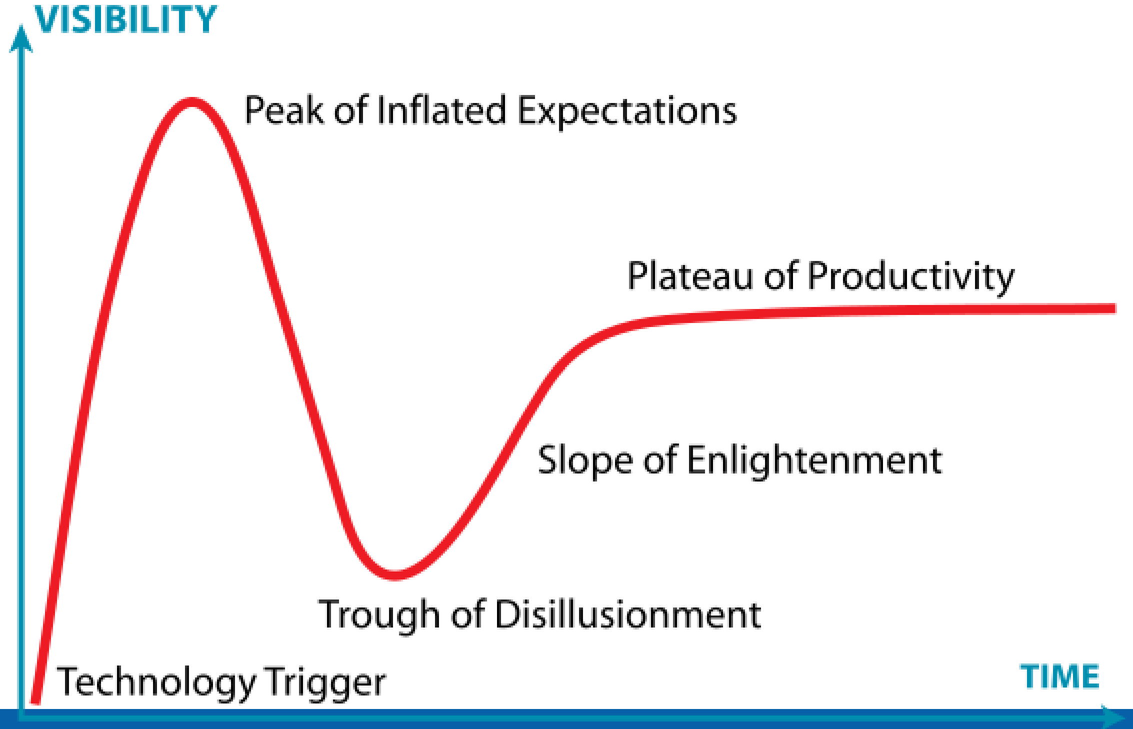
- ACI(D) properties of a transaction make it easier to ensure that shared memory programs are correct.
  - *Atomic*: each transaction either commits (it takes effect) or aborts (its effects are discarded).
  - *Consistent* (or *serializable*): they appear to take effect in a one-at-a-time order.
  - *Isolated* from other operations: the effects are not seen until the transaction has committed.
  - *(Durable*: their effects are persistent.)

Reasons for “I Hate TM”

- STM could be inefficient (most serious)
  - Improving rapidly, FUD, we were asked to address this
- TM will Never catch on, just use functional program
  - New programming style vs legacy
- Shared Mem is doomed, TM is evil because it makes Shared mem easier to use
- What concurrency software crisis? Nothing wrong with what we do today.
- Its too early
- TM still does not make your application parallel

Mission creep and Hype Cycle

- Now it is viewed as panacea for how hard it is to program multicore
- Can it help power consumptions?
- Use in embedded devices



Language support

- Programming Clojure
- Scala
- Haskell
- Perl
- Python
- Caml
- Java
- C/C++

Agenda

- STM, HTM, HybridTM
- **Birth of a specification**
- Design Goals
- Motivation for SG5 in C++ Standard
  - Use cases
  - Usability
  - Performance
- Language Constructs
  - Transactions, atomic and relaxed
  - Race-free semantics
  - Unsafe statements
  - Attributes
  - Transaction expressions and try blocks
  - Cancel
  - Exception handling
- SG5 Progress

Why do we need a TM language?

- TM requires language support
- Hardware here and now
- Multiple projects extend C++ with TM constructs
- Adoption requires common TM language extensions
- Draft specification of transactional language constructs for C++
  - 2008: Discussions by Intel, Sun, IBM started in July
  - 2009: Version 1.0 released in August
  - 2011: Version 1.1 fixes problems in 1.0, exceptions
  - 2012: Brought proposal to C++Std SG1; became SG5
  - 2013: close to wording for a C++ Technical Specification
- Today's talk: part motivation and part tutorial
- If time permits: part future specification

What is hard about adding TM to C++

- Conflict with C++ 0x memory model and atomics
- Support member initializer syntax
- Support C++ expressions
- Work with legacy code
- Structured block nesting
- Multiple entry and exit from transactions
- Polymorphism
- Exceptions

Agenda

- STM, HTM, HybridTM
- Birth of a specification
- **Design Goals**
- Motivation for SG5 in C++ Standard
  - Use cases
  - Usability
  - Performance
- Language Constructs
  - Transactions, atomic and relaxed
  - Race-free semantics
  - Unsafe statements
  - Attributes
  - Transaction expressions and try blocks
  - Cancel
  - Exception handling
- SG5 Progress

Design goals

Build on the C++11 specification
- Follow established patterns and rules
- “Catch fire” semantics for racy programs

Enable easy adoption
- Minimize number of new keywords
- Do not break existing non-transactional code

Restrict constructs to enable static error detection
- Ease of debugging is more important than flexibility

When in doubt, leave choice to the programmer
- Abort or irrevocable actions?
- Commit-on-exception or rollback-on-exception?

Agenda

- STM, HTM, HybridTM
- Birth of a specification
- Design Goals
- **Motivation for SG5 in C++ Standard**
  - Use cases
  - Usability
  - Performance
- Language Constructs
  - Transactions, atomic and relaxed
  - Race-free semantics
  - Unsafe statements
  - Attributes
  - Transaction expressions and try blocks
  - Cancel
  - Exception handling
- SG5 Progress

Overview

- **Use cases**: where is TM most useful?
- **Usability**: is TM easier than locks?
- **Performance**: is TM fast enough?

Use Cases

Locks are Impractical for Generic Programming=callback

Thread 1:
```
m1.lock();
m2.lock();
…
```

\+

Thread 2:
```
m2.lock();
m1.lock();
…
```

= *deadlock*

Easy. Order Locks.

Now let's get slightly more real:

What about Thread 1 +

A thread running `f()`*:*
```
template <class T>
void f(T &x, T y) {
  unique_lock<mutex> _(m2);
  x = y;
}
```

?

What locks does `x = y` acquire?

What locks does `x = y` acquire?

- Depends on the type `T` of `x` and `y`.
  - The author of `f()` shouldn't need to know.
    - That would violate modularity.
  - But lets say it's `shared_ptr<TT>`.
    - Depends on locks acquired by `TT`'s destructor.
    - Which probably depends on its member destructors.
    - Which I definitely shouldn't need to know.
    - But which might include a `shared_ptr<TTT>`.
      - Which acquires locks depending on `TTT`'s destructor.
      - Whose internals I definitely have no business knowing.
      - ...
- And this was for an unrealistically simple `f()`!
- We have no realistic rules for avoiding deadlock!
  - In practice: Test & fix?

```
template <class T>
void f(T &x, T y) {
  unique_lock<mutex> _(m2);
  x = y;
}
```

Transactions Naturally Fit Generic Programming Model

- Composable, no ordering constraints

f() implementation:
```
template <class T>
void f(T &x, T y) {
  transaction {
  x = y;
  }
}
```

Class implementation:
```
class ImpT
{
  ImpT& operator=(ImpT T&
rhs)
  {
    transaction {
      // handle assignment
    }
  }
};
```

Impossible to deadlock

The Problem

- **Popular belief:** *enforced locking ordering can avoid deadlock.*

- We show this is essentially impossible with C++ template programming.

- *Template programming is pervasive in C++. Thus, template programming needs TM.*

Don’t We Know This Already?

- Perhaps, but impact has been widely underestimated.
  - Templates are everywhere in C++.
- Move TM debate away from performance; focus on convincingly correct code.
- Relevant because of C++11 and SG5.
- Generic Programming Needs Transactional Memory by Gottschlich & Boehm, Transact 2013

Conclusion

- Given C++11, generic programming needs TM more than ever.

- To avoid deadlocks in ***<u>all</u>*** generic code, even those with irrevocable operations, we need (something like) relaxed transactions.

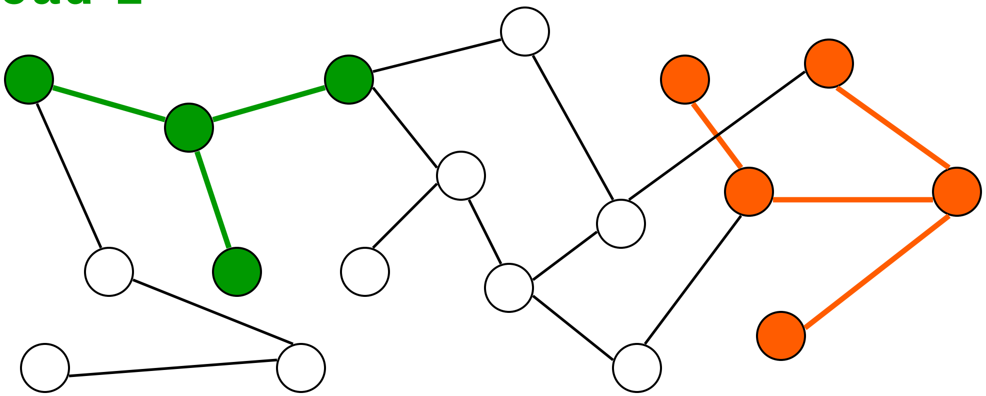
TM Patterns and Use Cases

- Top four uses cases:
  1. Irregular structures with low conflict frequency
  2. Low conflict structures with high read-sharing and complex operations
  3. Read-mostly structures with frequent read-only operations
  4. Composable modular structures and functions

Irregular Structures

- Irregular structures with low conflict frequency
  - E.g., graph applications (minimum spanning forest sparse graph, VPR and FPGA)
  - Advantages: concurrency and ease of deadlock-avoidance, ease of programming

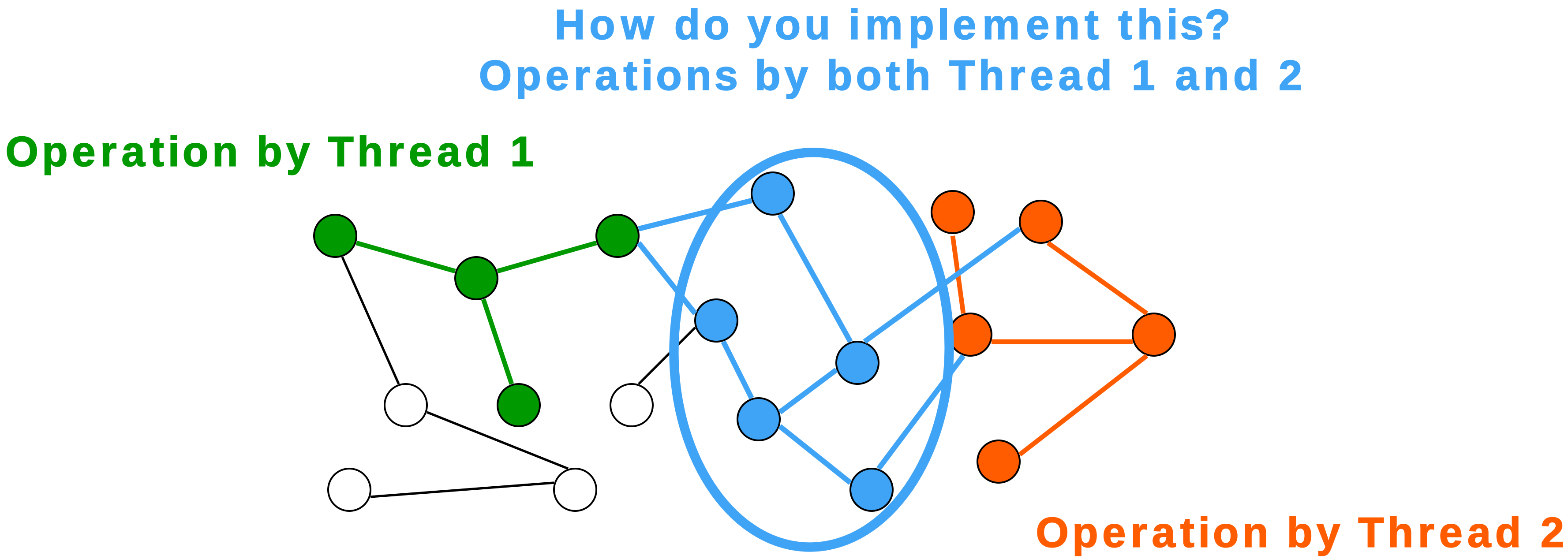
Operation by Thread 1



Operation by Thread 2

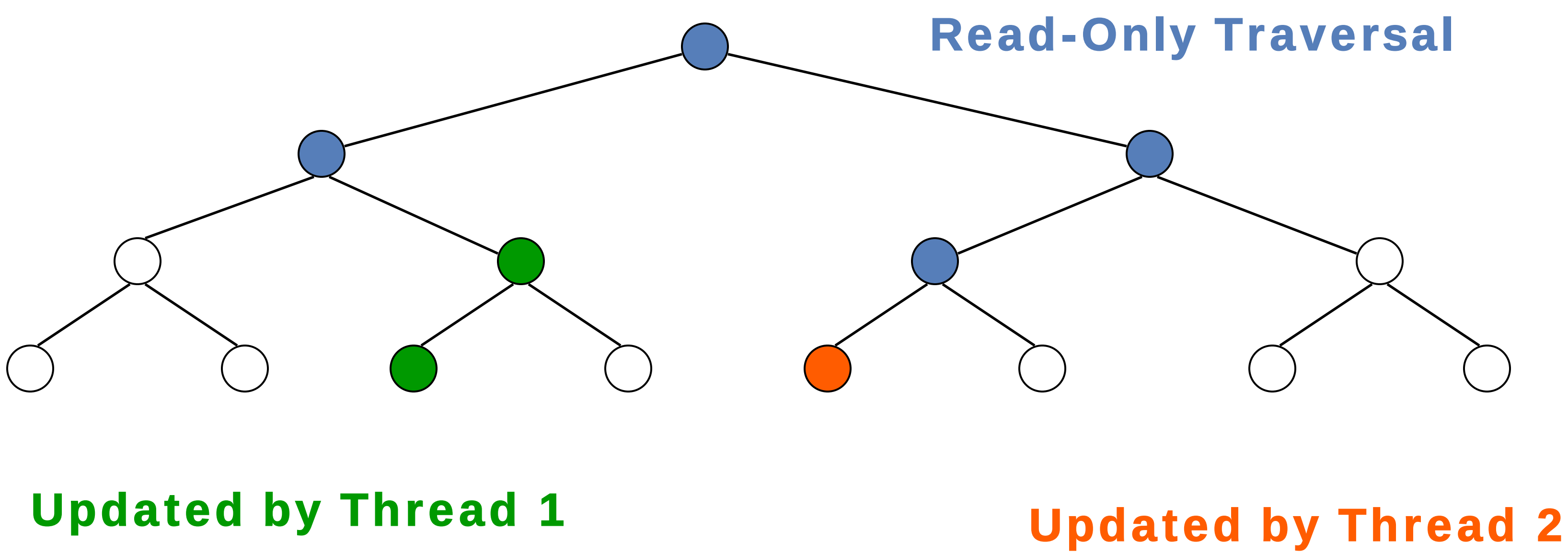
Why Not Locks?

- If conflicts arise, fine-graining locking can lead to deadlocks or degraded performance



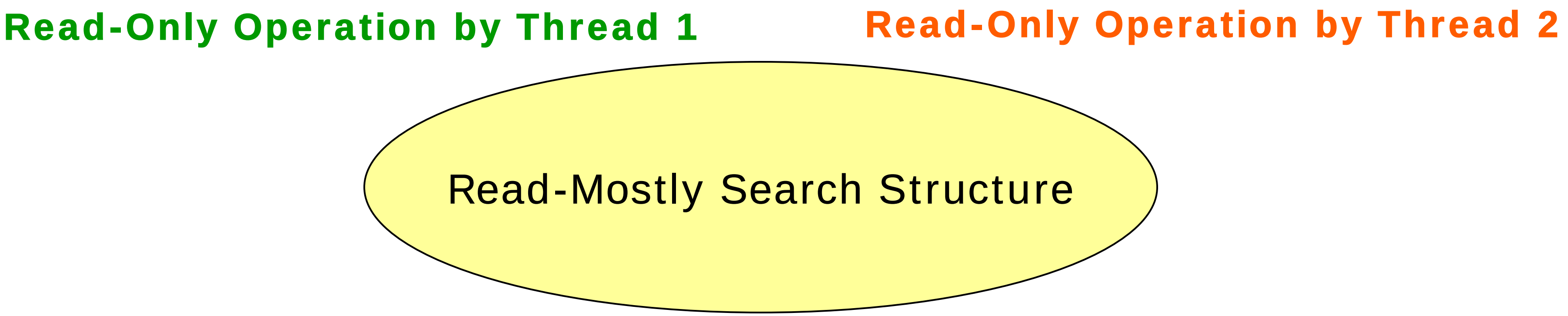
Low Conflict Structures

- Low conflict structures with high read-sharing and complex operations
  - E.g. red-black trees, AVL trees
  - Advantages: ease of parallelization, high concurrency, low cache coherence traffic, ease of programming



Read-Mostly Structures

- Read-mostly structures with frequent read-only operations
  - E.g. search structures
  - Advantages: high concurrency, read-only operations avoid writing (avoid unnecessary cache coherence traffic)



Composition / Modularity

- Arbitrarily composable modular structures and functions
  - Advantages: modular design, code maintainability, ease of programming (e.g., using STL)

```
__transaction {
  // Search an arbitrary structure A for an item with an arbitrary key K
  // If found, remove that item (X) from A
  X = remove(A,K);
  if (X != NULL)
  {
     // Depending on X’s value, insert X in an arbitrary structure B
     B = f(X->Value);
     insert(B,X);
  }
}
```

Usability

Two User Studies

- Is Transactional Programming Actually Easier?
  - Chris Rossbach, Owen Hofmann, Emmett Witchel
  - 3-year study of undergrad class (237 students)
  - presented at PPoPP 2010
- A Study of TM vs. Locks in Practice
  - Victor Pankratius, Ali-Reza Adl-Tabatabai
  - 6 groups, each with 2 Masters students
  - presented at SPAA 2011

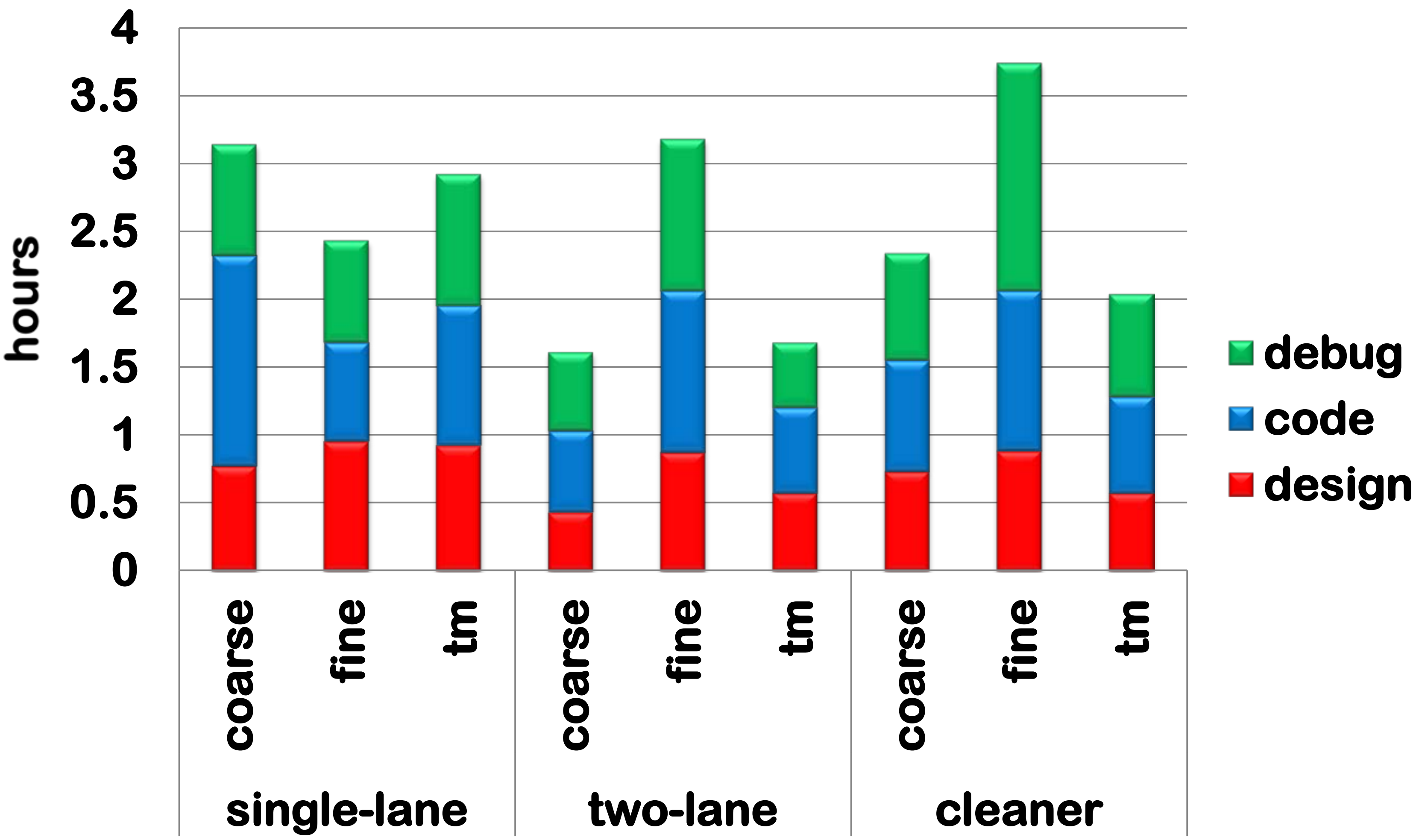
Is Transactional Programming Actually Easier?

- "Sync-gallery" programming assignment
  - part of undergrad OS course
    - 2 sections in each of 3 semesters, each a year apart
    - 237 students total
  - assignment had 3 variants (see next slide)
  - each student implemented each variant 3 ways
    - coarse-grained locking, always done first
    - fine-grained locking
    - TM (library-based support only)
    - randomly assigned which of fine-grained locking or TM-based to implement first

Sync-gallery assignment

- “Rogues” shoot paint balls in “lanes” at a gallery
  - 2 rogues (one shoots red, the other blue), 16 lanes
- Four properties
  - only one rogue can shoot in a lane at a time
  - must shoot in “clean” lane
  - clean all lanes when there are no more clean lanes
  - only one thread cleaning at a time (no concurrent shooting)
- Three variants
  - rogue reserves one lane at a time, cleans all lanes if it dirties the last lane
  - same as above, except rogue reserves two lanes at a time
  - all cleaning by separate cleaner thread; coordinate via condition variable

Development Effort: year 2



Qualitative preferences: Y2

Best Syntax

Ranking	1	2	3	4
Coarse	**62%**	30%	1%	4%
Fine	6%	21%	**45%**	**40%**
TM	26%	**32%**	19%	21%
Conditions	6%	21%	29%	**40%**

Easiest to Think about

Ranking	1	2	3	4
Coarse	**81%**	14%	1%	3%
Fine	1%	**38%**	30%	29%
TM	16%	**32%**	30%	21%
Conditions	4%	14%	**40%**	**40%**

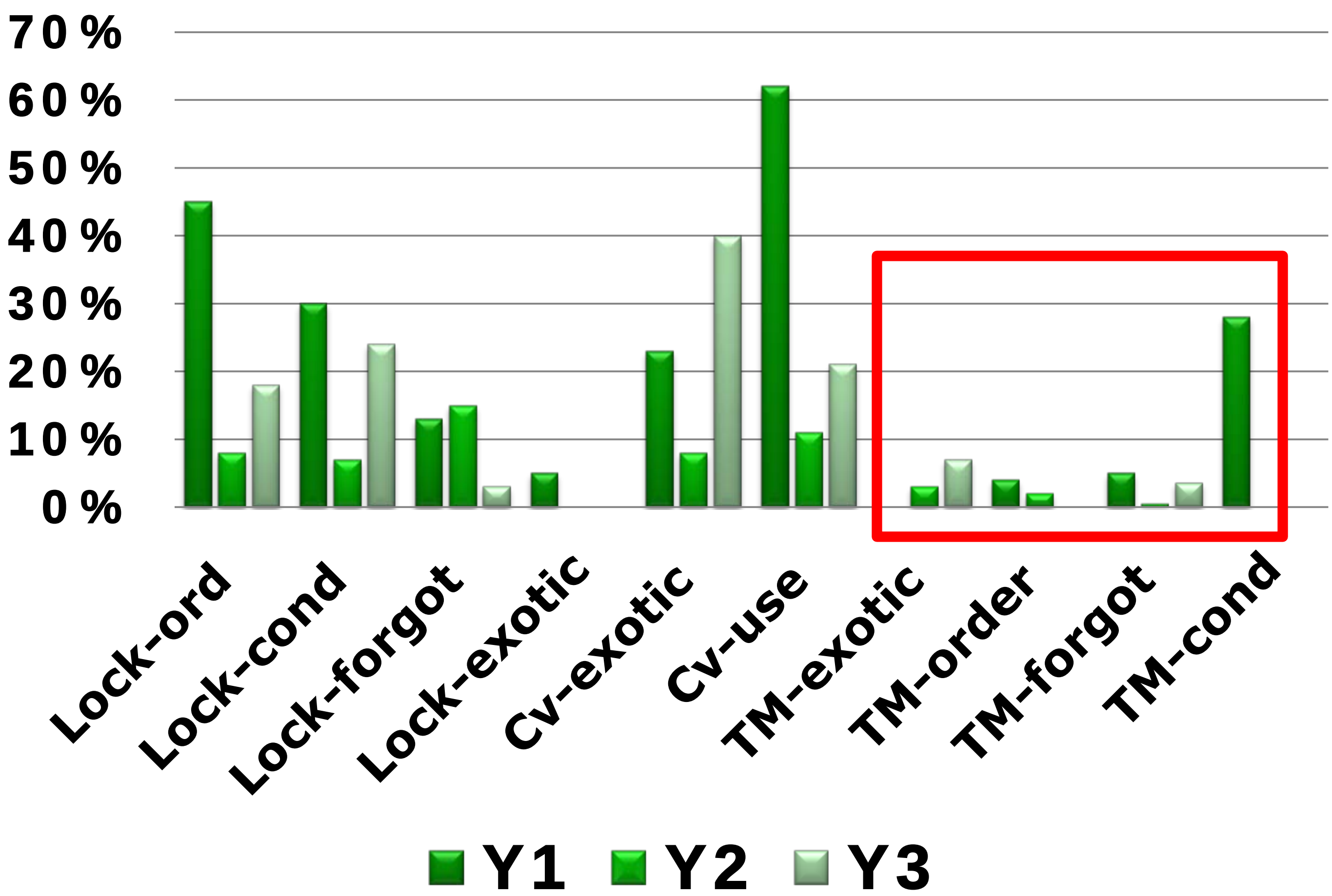
(Year 2)

Analyzing Programming Errors

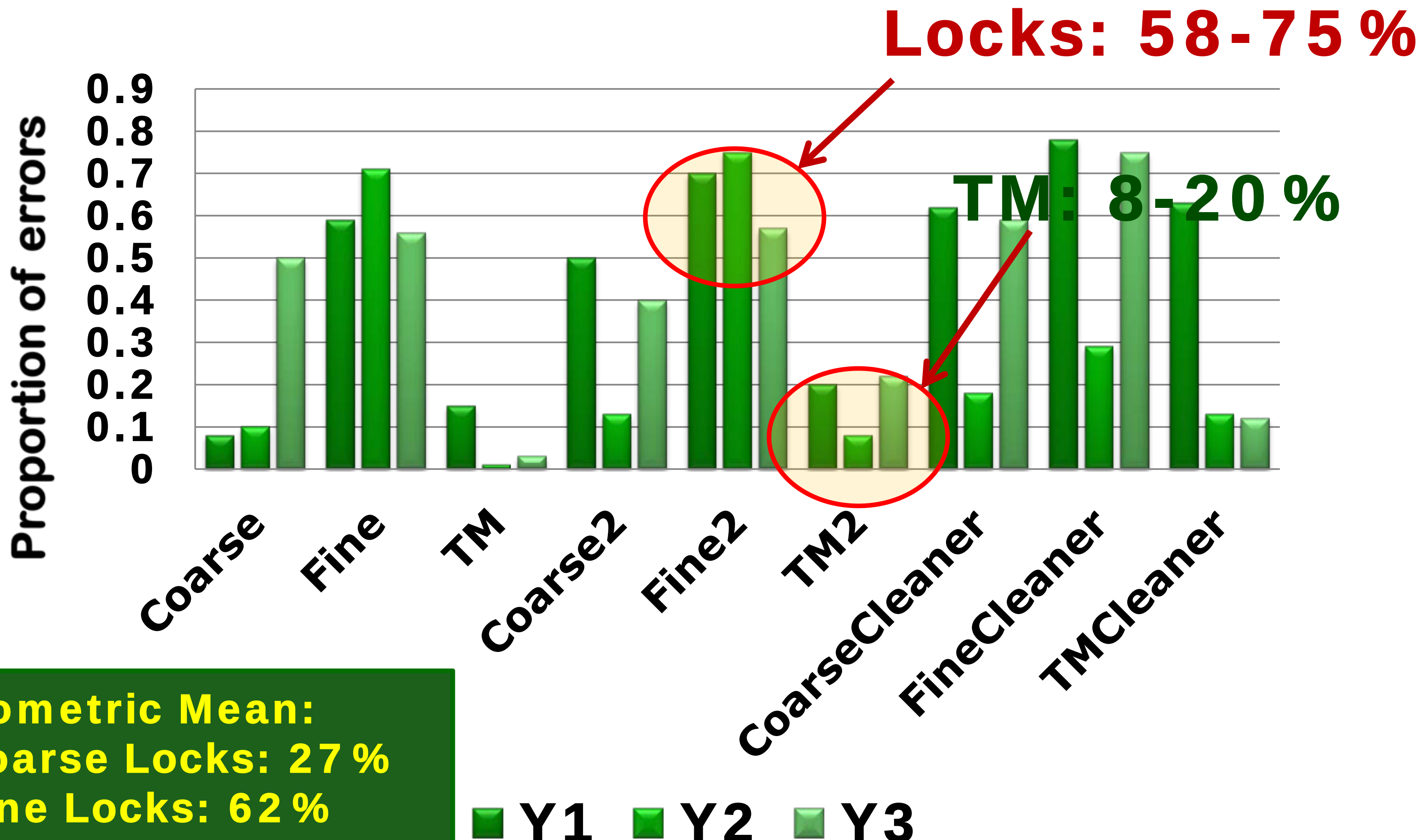
Error taxonomy: 10 classes

- **Lock-ord:** lock ordering
- **Lock-cond:** checking condition outside critical section
- **Lock-forgot:** forgotten synchronization
- **Lock-exotic**: inscrutable lock usage
- **Cv-exotic:** exotic condition variable usage
- **Cv-use:** condition variable errors
- **TM-exotic:** TM primitive misuse
- **TM-forgot:** forgotten TM synchronization
- **TM-cond**: checking conditions outside critical section
- **TM-order:** ordering in TM

Error Rates by Defect Type

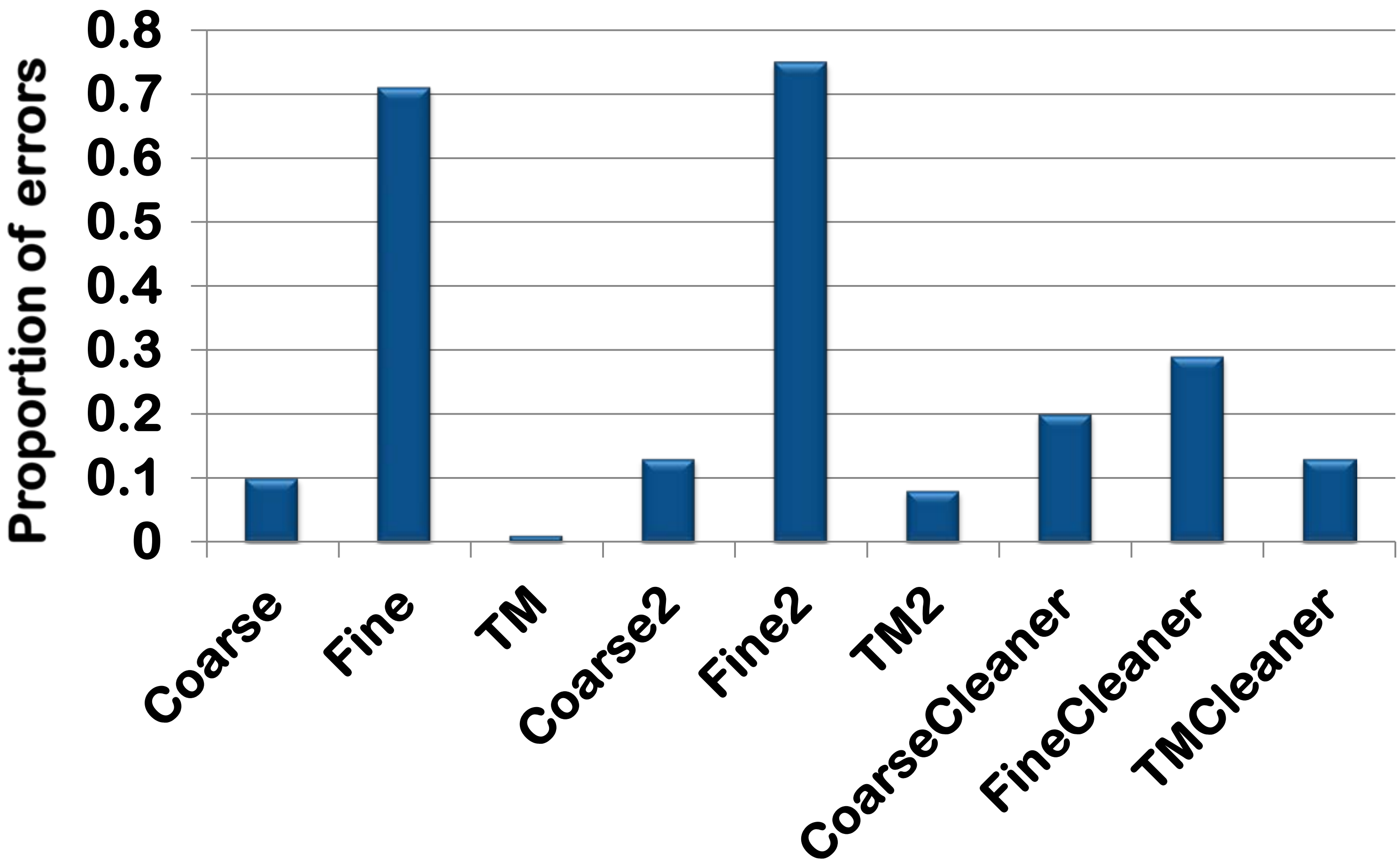


Overall Error Rates



Geometric Mean:
- **Coarse Locks: 27%**
- **Fine Locks: 62%**
- **TM: 10%**

Overall Error Rates: Year 2



Comments and conclusions

- TM problems
  - lack of documentation/tutorial
  - initial syntax of library-based TM
    - better in years 2/3 with different TM library
- Students found
  - TM harder than coarse-grained locking
  - TM easier than fine-grained locking and condition vars.
- Much fewer errors for TM than for locking

A Study of Transactional Memory vs. Locks in Practice

- “Explorative case study”
  - Broad scope
  - Less control, more realism
  - Lessons learned on a case-by-case basis
  - Programmed a desktop search engine

The Project: Parallel Desktop Search Engine

- 15 week project
- 12 subjects (Master's students)
  - Prior to project, same training for everyone
    (Parallel programming, locks / Pthreads, TM using Intel's STM compiler)
  - Randomly created 6 teams (2 students each)
    - 3 teams randomly assigned to use locks
    - 3 teams TM + Phreads
  - All using the same spec for indexing and search
- Collecting evidence
  - Code, svn, time records, weekly interviews, student diaries, notes, post-project questionnai observations

Code

- Average LOC about the same
- TM teams have fewer LOC with parallel constructs (2%-5% vs. 5%-11%)

	Locks Teams			TM Teams		
	L1	L2	L3	TM1	TM2	TM3
Total Lines of Code (excl. comments, blank lines)	**2014**	**2285**	**2182**	**1501**	**2131**	**3052**
	avg: 2160 stddev: 137			**avg: 2228 stddev: 780**		
LOC pthread*	157 8%	261 11%	120 5%	17 1%	23 1%	12 0%
LOC tm_*	0	0	0	36 2%	22 1%	139 5%
LOC with paral. constr. (pthread* + tm_*)	157 8%	261 11%	120 5%	53 4%	45 2%	151 5%
	avg: 179 stddev: 73			**avg: 83 stddev: 59**		

Code

- Locking programs more complex than TM
  - code inspections revealed thousands of locks
- TM teams combined transactions and locks
  - TM2: one lock to protect a large critical section containing I/O
  - TM3: two semaphores for producer-consumer synchronization
- All locks teams used condition variables, but none of the TM teams did
- Sync constructs rarely lexically nested

Code inspections with compiler experts at Intel

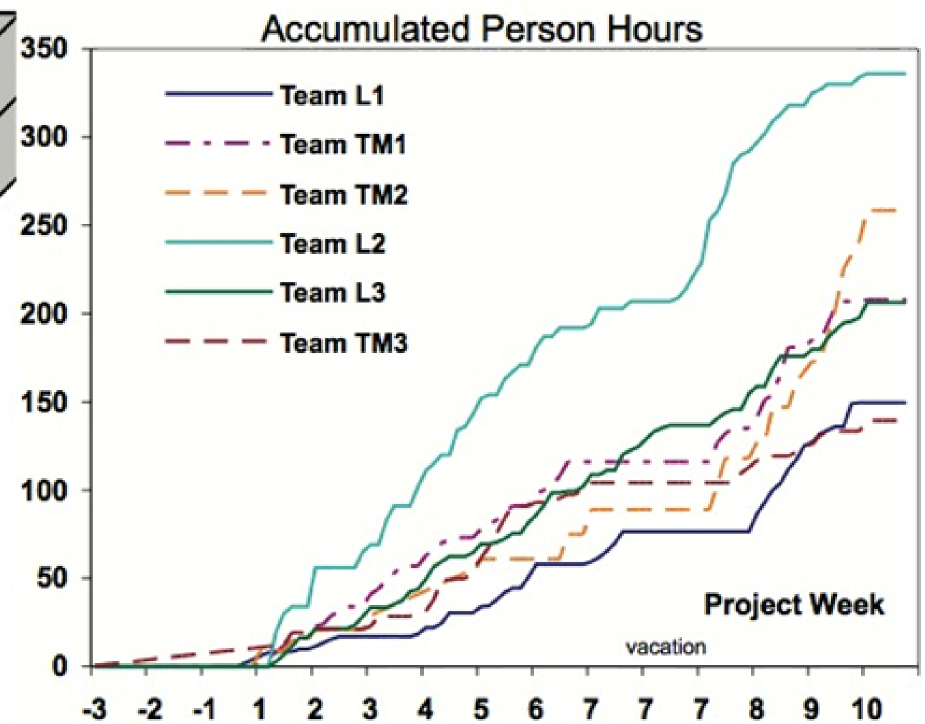
- Locks programs need fine-grained locking for scalability, but many locks complicate program understanding
  - L2: 1600 locks, L3: 80 locks, L1: 54 locks
  - L2 the only locking program to scale on indexing
- TM teams used locks and transactions to perform producer-consumer synchronization, perform I/O, and optimize access to immutable data
- Double-checked locking patterns in both locks and TM teams
  - Attempt to optimize performance
- Common mistake: unprotected reading of shared state. Exception: L1

Programming Effort

~14% difference in total programming effort in favor of TM

Total Effort (Person Hours)

	Reading	Search for Libraries	Design	Implementation	Additional Experiments	Testing	Debugging	Other	Total
Team L1	6	3	9	80	10	14	29	0	151
Team L3	24	1	17	72	7	52	16	19	208
Team L2	29	12	14	196	12	21	48	2	334
Team TM3	18	4	12	55	6	18	19	9	141
Team TM1	7	6	33	74	18	38	22	10	208
Team TM2	6	6	21	139	12	39	38	0	261
sum all	**90**	**32**	**106**	**616**	**65**	**182**	**172**	**40**	**1303**
	7%	2%	8%	47%	5%	14%	13%	3%	100%
sum L	**59**	**16**	**40**	**348**	**29**	**87**	**93**	**21**	**693**
	9%	2%	6%	50%	4%	13%	13%	3%	100%
sum TM	**31**	**16**	**66**	**268**	**36**	**95**	**79**	**19**	**610**
	5%	3%	11%	44%	6%	16%	13%	3%	100%
sum L - sumTM	*28*	*0*	*-26*	*80*	*-7*	*-8*	*14*	2	83



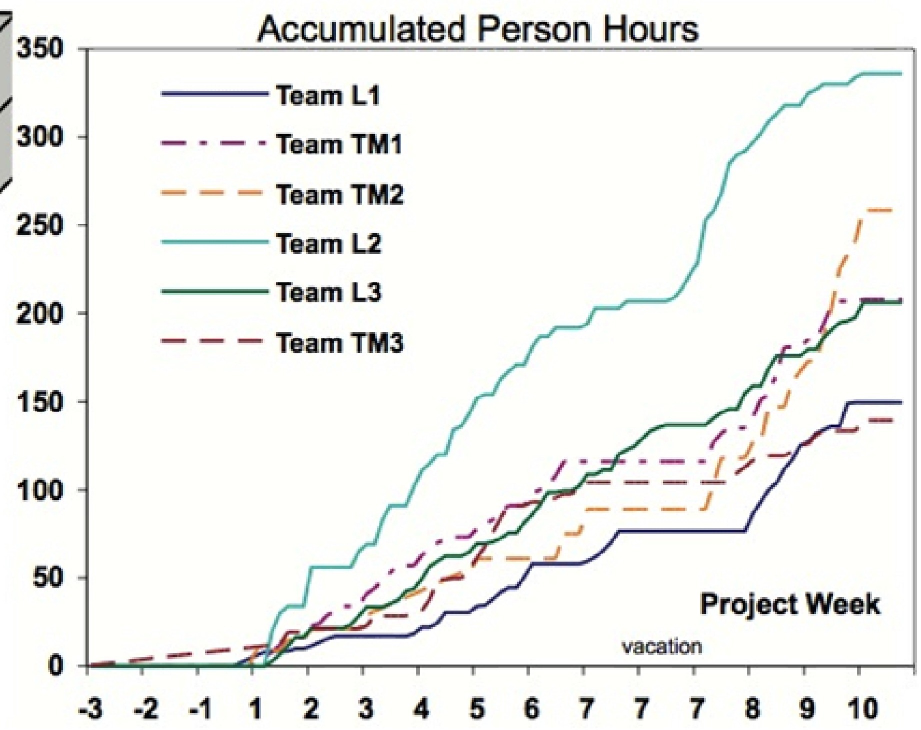
Less for TM

Increase for TM teams in last weeks: Refactoring transactions, performance problems, experiments

Programming Effort

Total Effort (Person Hours)

	Reading	Search for Libraries	Design	Implementation	Additional Experiments	Testing	Debugging	Other	Total
Team L1	6	3	9	80	10	14	29	0	151
Team L3	24	1	17	72	7	52	16	19	208
Team L2	29	12	14	196	12	21	48	2	334
Team TM3	18	4	12	55	6	18	19	9	141
Team TM1	7	6	33	74	18	38	22	10	208
Team TM2	6	6	21	139	12	39	38	0	261
sum all	**90**	**32**	**106**	**616**	**65**	**182**	**172**	**40**	**1303**
	7%	2%	8%	47%	5%	14%	13%	3%	100%
sum L	**59**	**16**	**40**	**348**	**29**	**87**	**93**	**21**	**693**
	9%	2%	6%	50%	4%	13%	13%	3%	100%
sum TM	**31**	**16**	**66**	**268**	**36**	**95**	**79**	**19**	**610**
	5%	3%	11%	44%	6%	16%	13%	3%	100%
sum L - sumTM	*28*	*0*	*-26*	*80*	*-7*	*-8*	*4*	*2*	*83*



Debugging segfaults

- Locks teams: 55 hours (59%) of debugging time
- TM teams: 23 hours (29%) of debugging time

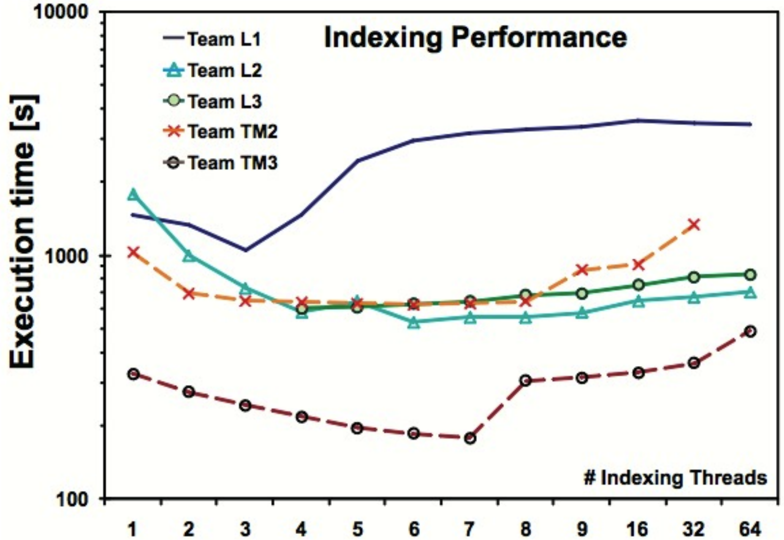
→ Influenced by LOC containing parallel constructs

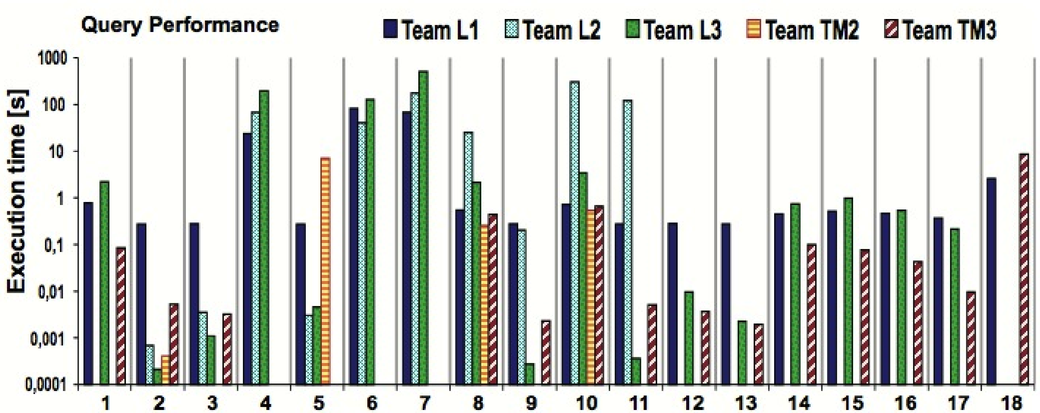
Parallelization Progress

- TM allowed teams to think more sequentially
  - spent 50% less time as the locks teams on writing parallel code
  - Hours spent on sequential code versus parallel code
  - Time lag between the first day of work on sequential code and the first day of work on parallel code
    - (L1 :1 day, L2: 13 days, L3: 19 days) vs. (TM3, 19 days, TM2: 23 days, TM1: 29 days)
  - Yet TM3 had first working parallel version, even though they subjectively believed they advanced slowly

- By project deadline
  - L1 had performance problems, skipped performance tests
  - L2 did not finish performance tests
  - L3 discovered a new concurrency bug (winner for locks)
  - TM1 fails on benchmark
  - TM2 reasonable performance
  - TM3 excellent (winner for TM)

Performance





Query type

1 one frequent word	6 frequent text passage (3 words)	11 wildcard rare (*word)	16 exclusion (1 frequent word)
2 one rare word	7 rare text passage (3 words)	12 AND frequent (2 words)	17 exclusion (1 rare word)
3 one random word	8 wildcard frequent (word*)	13 AND rare (2 words)	18 AND four characters with wildcards
4 frequent text passage (2 words)	9 wildcard rare (word*)	14 AND frequent (3 words)	
5 rare text passage (2 words)	10 wildcard frequent (*word)	15 AND rare (3 words)	

- TM3 outperforms on indexing performance and most teams on query performance

→ Counterexample that TM performance need not be bad in practice

Performance

Is TM Fast Enough?

- Many different STMs with different goals (and different guarantees)
  - TL2: baseline state-of-the-art
  - TinySTM: added safety guarantees (opacity)
  - NOrec: generalized support of many features
  - InvalSTM: contention-heavy programs
  - SkySTM: scalable to upwards of 250 threads

- How to choose?
  - Use adaptive algorithm (Wang et al., HiPEAC'12)
  - ***Change TM without changing client code***

Commercial Hardware TMs

- Azul Systems' HTM (phased out?)
- AMD ASF (unknown status)
- Sun's Rock (cancelled)
- IBM's Blue Gene/Q (2011)
- Intel's TSX (code named Haswell) (2012)
- IBM's zEC12 (2012)

- HTM will only improve existing STM performance

Commercial/OS Compilers

- Sun Studio (for Rock)
- Intel STM
- IBM AlphaWorks STM (for BG)
- GNU 4.7
- IBM xlC z/OS v1R13 compiler

Intel 12.2 and GNU 4.7 support

- Both based on Draft C++ TM spec
- Intel is based on V1.0, but has many extensions
- GNU is based on V1.1
  - See slide on Draft 1.1 addition for differences
- Both use a form of Intel TM ABI V1.1 2006/05/06
  - GNU does not implement all of the ABI (mostly missing the Intel TM extensions)

Intel C++ STM Prototype Edition 4.0

- All of Draft 1.0 + extensions in support of the Intel ABI
  - So no block support, uses GCC attributes for atomic and relaxed transactions
- Intel extensions:
  - New language constructs __tm_atomic { ...} else { ... } and __tm_wavier {...}
  - New function annotations __tm_safe, and tm_wrapping with support for registering commit and undo handlers for writing advanced transactional libraries
  - Transactional C++ new/delete, constructor and destructor support
  - TM annotation for template classes
  - Support for abort-on-exception semantics with explicit _tm_abort throw <exception>
  - New compiler and runtime optimizations
  - Support for transactional C++ STL library

GNU 4.7

- All of Draft 1.1 except
  - No support for _transaction_cancel throw, no rollback on exceptions
    - Commit on throw is the default
  - No checking of consistency of function attributes
  - Not sure if it fully supports templates, they never checked

IBM xlC zSeries V1R13

- IBM XL C/C++ Compiler Maximizes zEC12's Transactional Execution Capabilities by Marcel Mitran and Visda Vokhshoori,
- Offers Low level transactional library functions
  - Enables begin, end, abort, nesting depth

Real-World STM Application

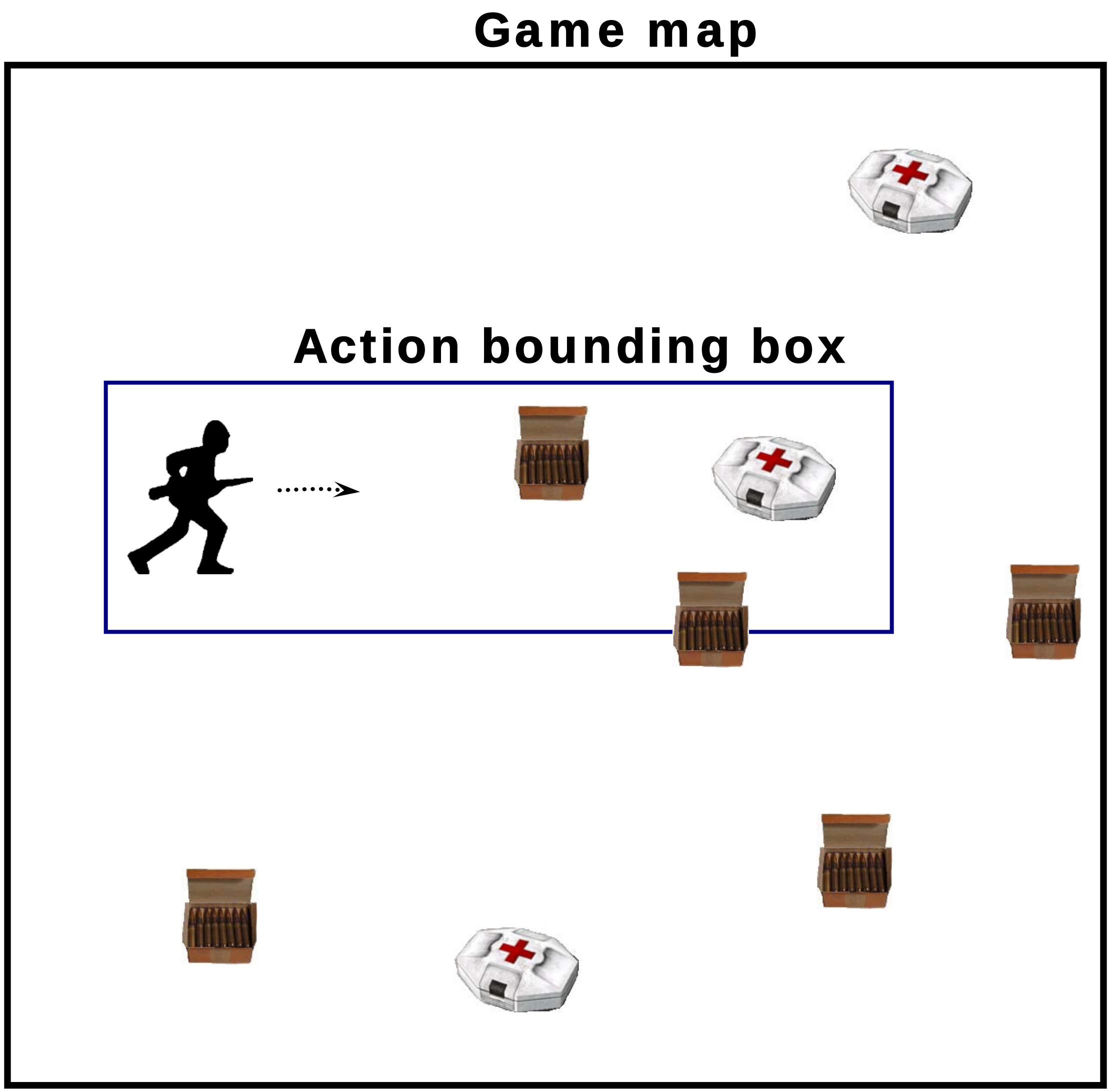
- Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games
  - Daniel Lupei, Bogdan Simion, Don Pinto, Mihai Burcea, Matthew Misler, William Krick, Cristiana Amza
  - application: SynQuake, simulates Quake battles
  - used software-only TM (STM)
  - presented at EuroSys 2010

Multiplayer games

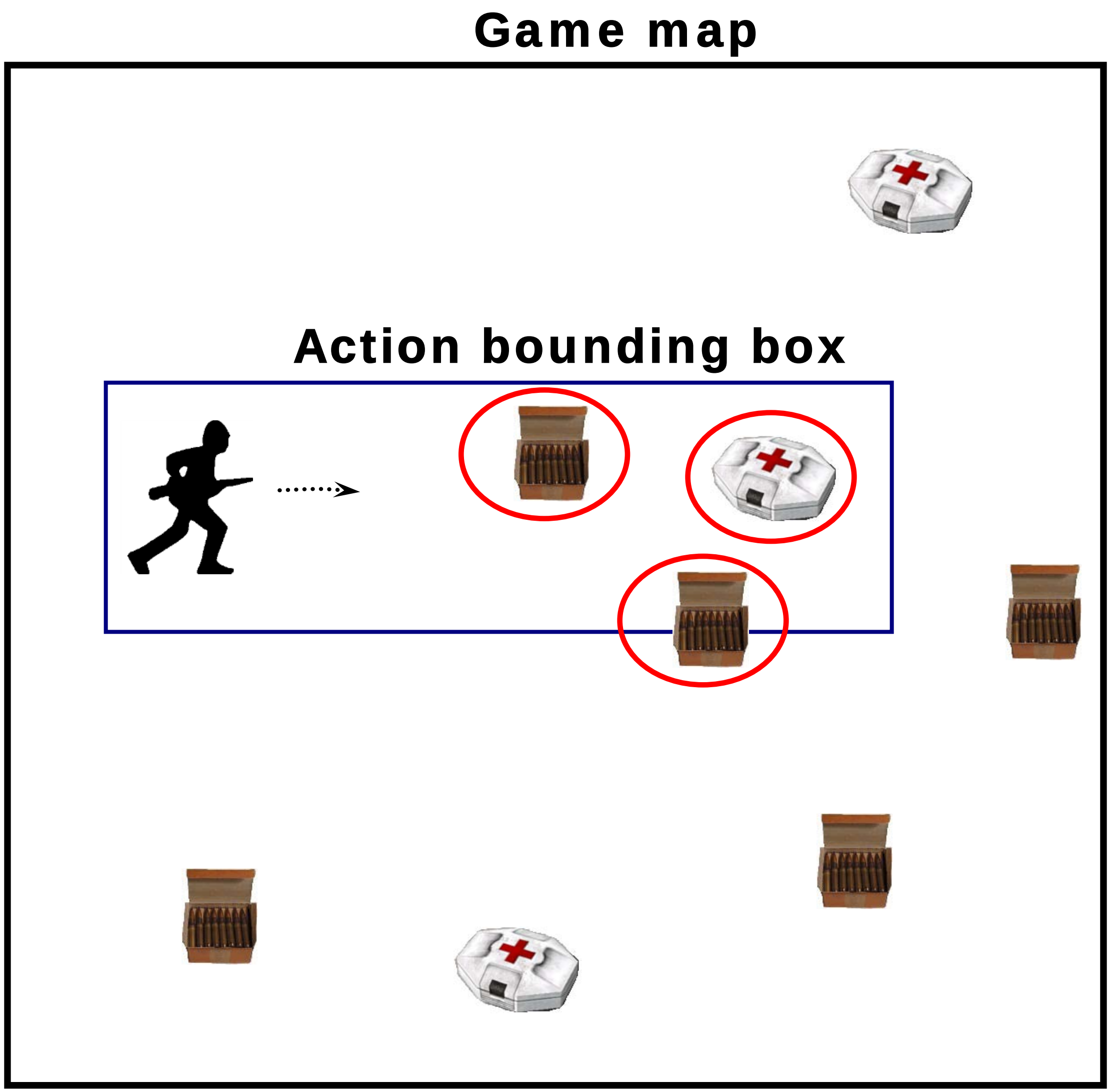
- More than 100k concurrent players

Game server is the bottleneck

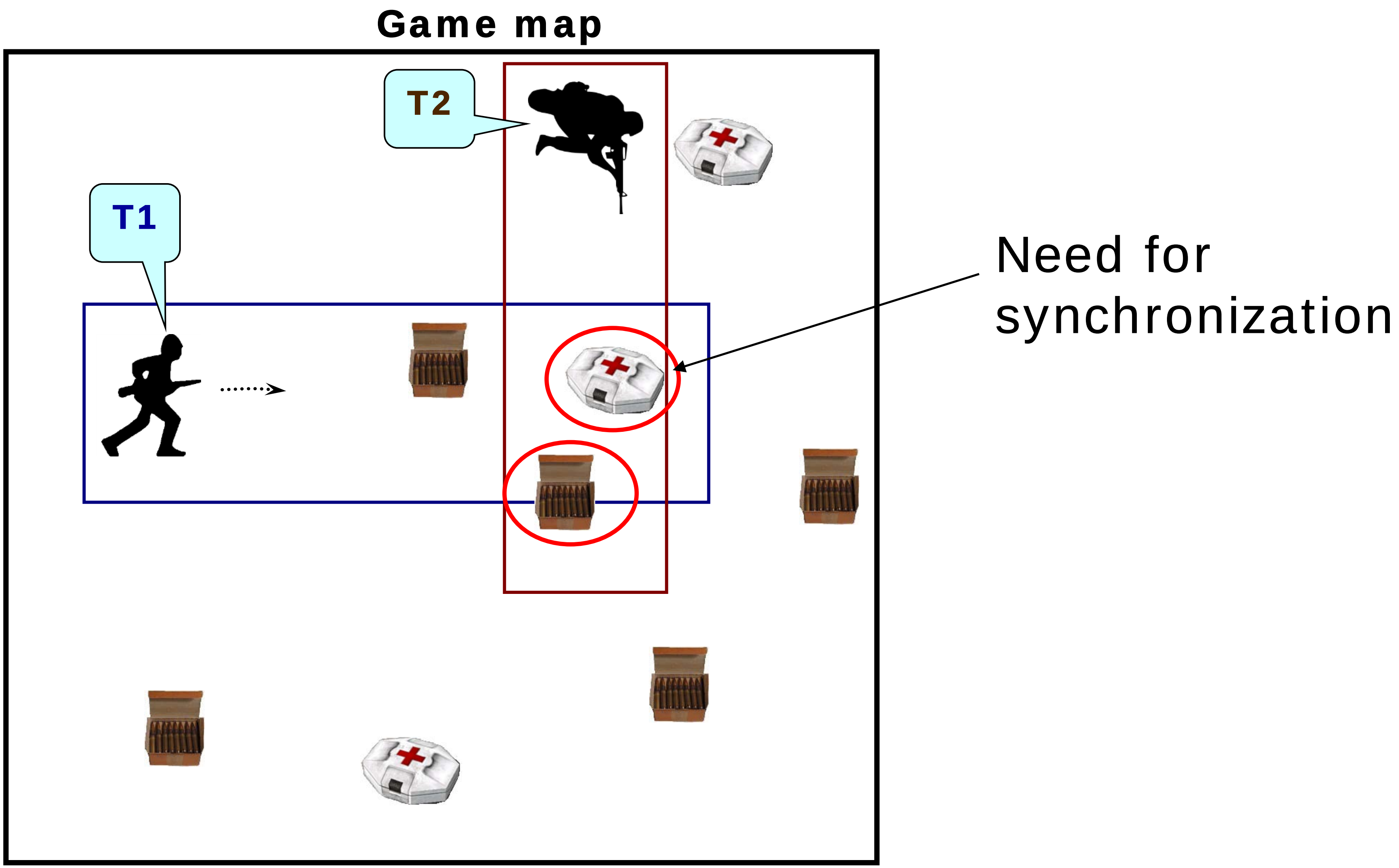
Game interactions



Collision detection



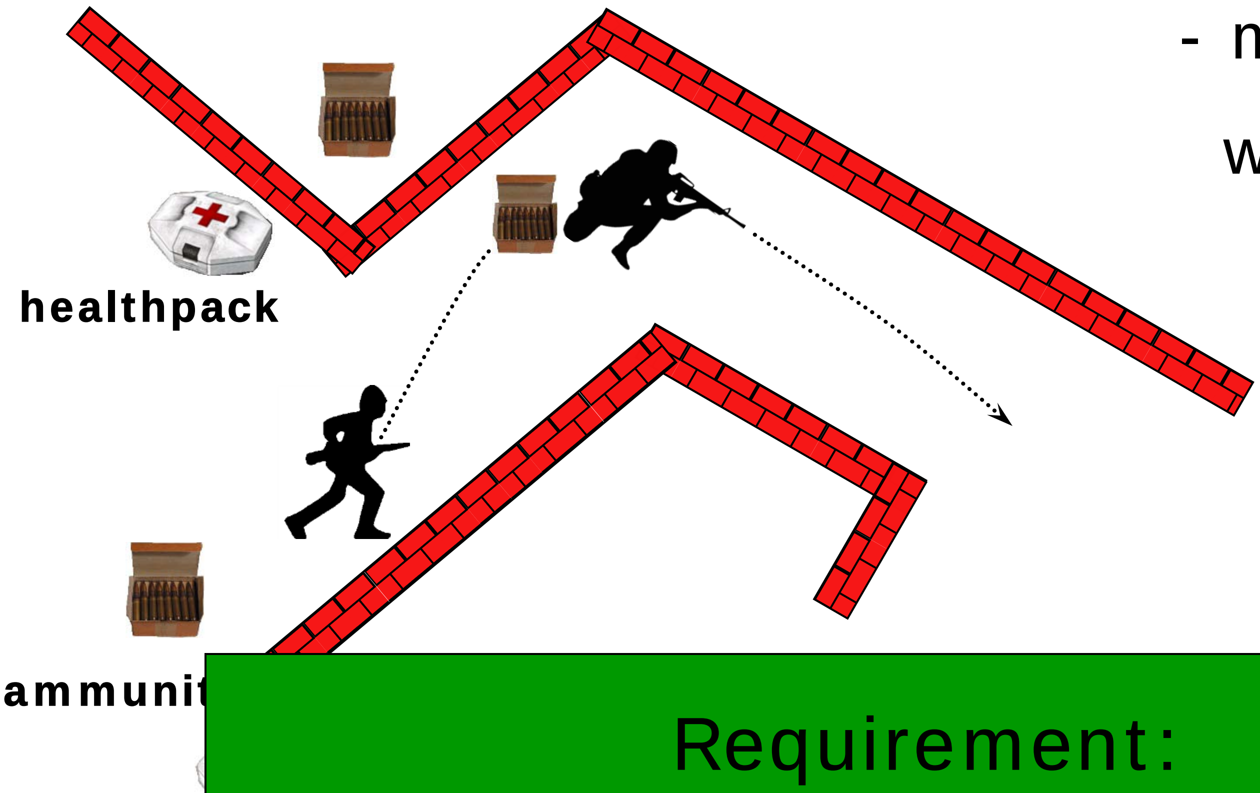
Conflicting player actions



Player actions

Compound action:

- move, charge weapon and shoot



Requirement:
consistency and atomicity
of whole game action

Conservative locking

GAME ACTION

Lock 1, Lock 2, Lock3

Subaction 1

Subaction 2

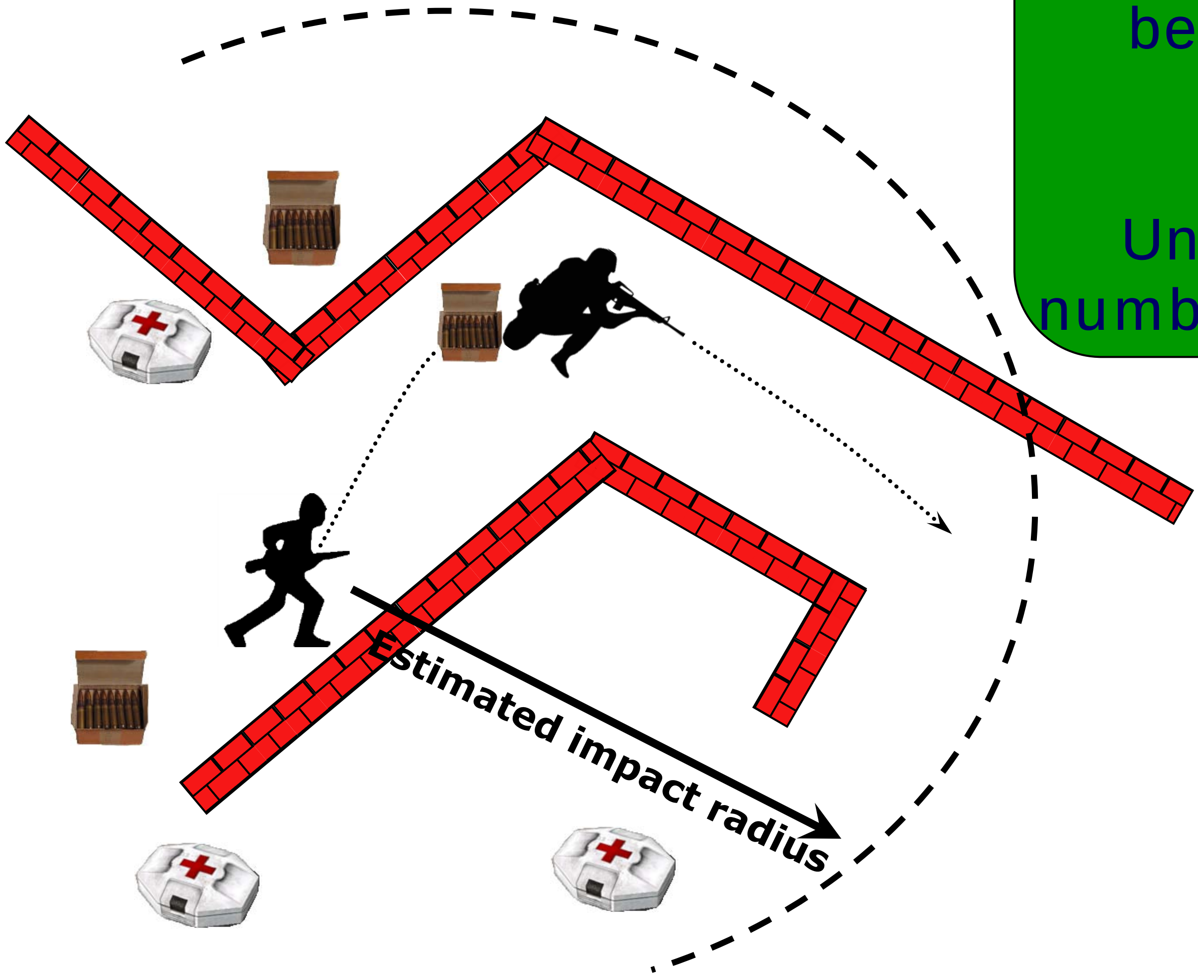
Subaction 3

Unlock 1,2,3

Conservatively acquire all locks at beginning of action

Problem 1:
Unnecessarily long conflict duration

Conservative locking



Conservative estimate of impact range at beginning of action

Problem 2:
Unnecessarily high number of locked objects

Fine-grained locking?

GAME ACTION

Lock 1

Subaction 1

Unlock 1

Lock 2

Subaction 2

Unlock 2

Lock 3

Subaction 3

Unlock 3

Not possible !

Problem:

- No atomicity for whole action

Fine-grained locking?

GAME ACTION

Lock 1

Subaction 1

Lock 2

Subaction 2

Lock 3

Subaction 3

Unlock 1, 2, 3

Not possible !

Problem:
- Deadlocks
- Inconsistent view

STM

- Alternative parallelization paradigm
  - Implement game actions as transactions
  - Track accesses to shared and private data
  - Conflict detection and resolution
- Automatic *consistency and atomicity*
  - Transaction commits if no conflict
  - Transaction rolls back if conflict occurs

STM - Synchronization

GAME ACTION

BEGIN Transaction

Subaction 1

Subaction 2

Subaction 3

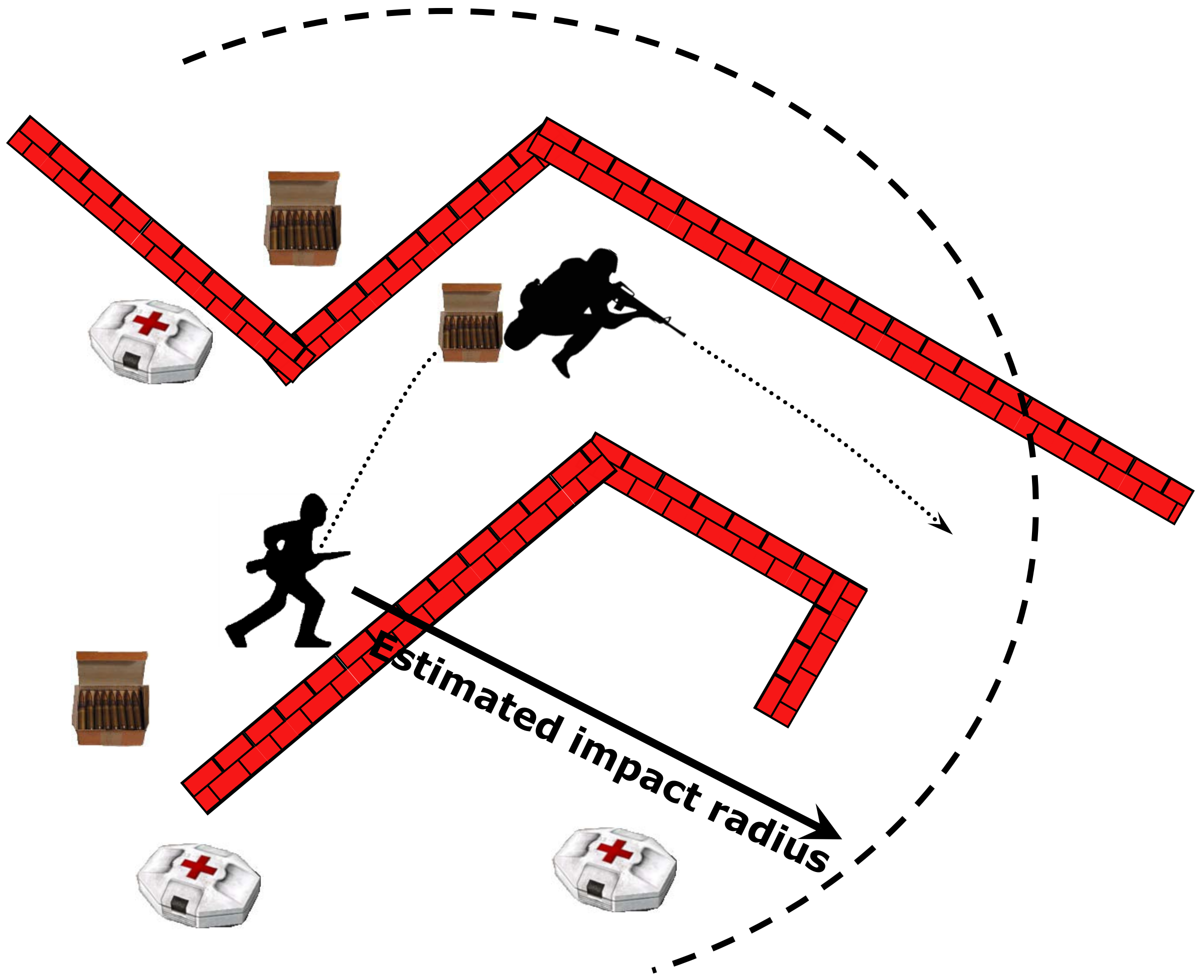
COMMIT Transaction

Problems solved:

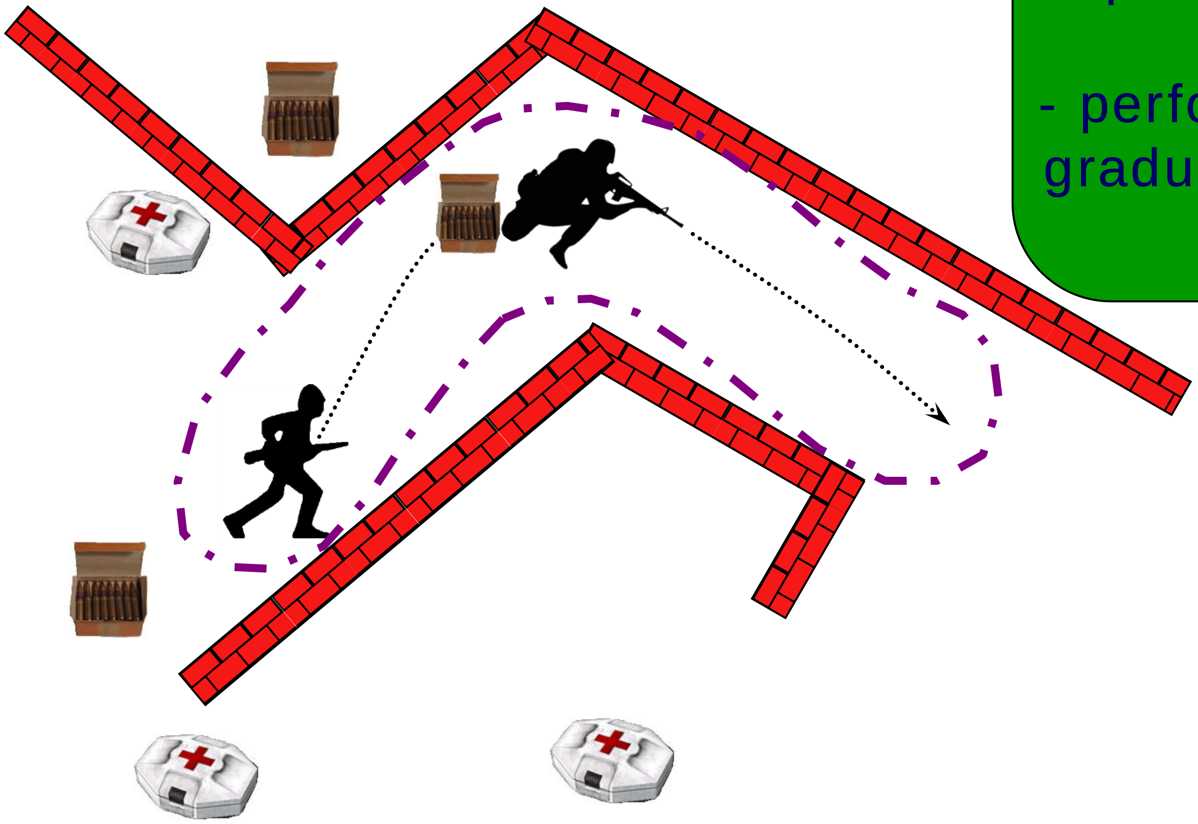
- Deadlocks
- Atomicity

Handled automatically

STM - Synchronization



STM - Synchronization



Collision detection optimized:

- split action into subactions

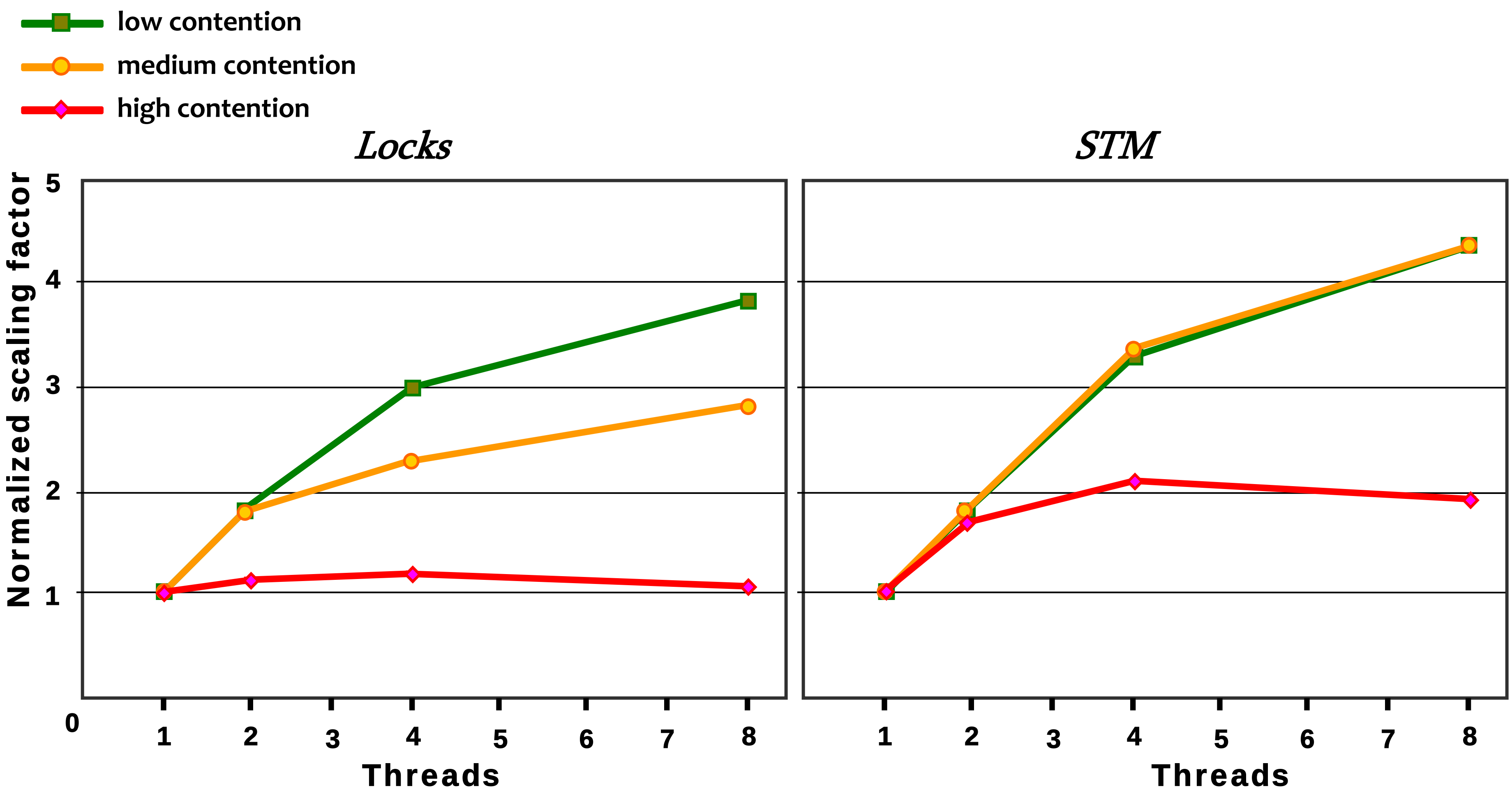
- perform collision detection gradually for each subaction

Experimental Results

- Test scenarios:
  - 1 – 8 quests, short/long range actions
- Performance comparison
  - Locks vs. STM scaling and performance
  - Influence of load balancing on scaling

Scalability

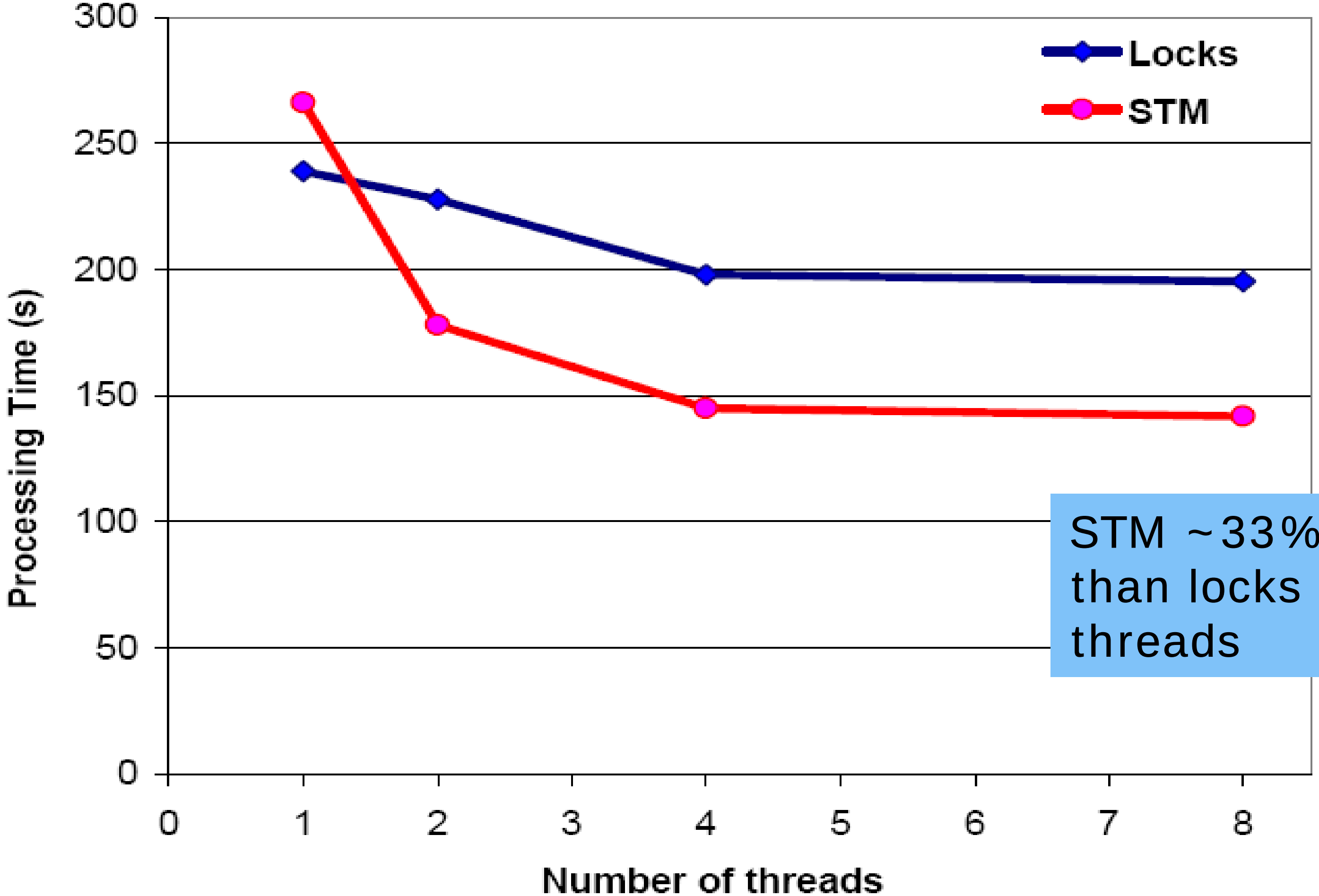
8 core machine



STM scales better in all 3 contention scenarios

Processing Times

Medium contention



STM ~33% faster than locks for 4-8 threads

Conclusions

- TM naturally aligns with generic programming
- Many problems are well-suited for TM
- Early studies show TM to be easy to program and less buggy than locks
- Software-only TM can outperform locks

Agenda

- STM, HTM, HybridTM
- Birth of a specification
- Design Goals
- Motivation for SG5 in C++ Standard
  - Use cases
  - Usability
  - Performance
- **Language Constructs**
  - Transactions, atomic and relaxed
  - Race-free semantics
  - Unsafe statements
  - Attributes
  - Transaction expressions and try blocks
  - Cancel
  - Exception handling
- SG5 Progress

Language constructs in a nutshell

3 constructs for transactions

1. Compound Statements
2. Expressions
3. Blocks: funtion-try-blocks, constructor-initializer

2 Keywords for different types of TX

__transaction_atomic [[[outer]]] [[[noexcept]]] <compound-statement>

__transaction_relaxed [[[noexcept]]] <compound-statement>

1 keyword only applies to atomic TX

__transaction_cancel [[[outer]]] [throw (<expr>)]

4 Function attributes

transaction_safe

transaction_unsafe

transaction_callable

transaction_may_cancel_outer

Transaction statement

- All 3 constructs support 2 forms

```
__transaction {x++;}
__transaction_atomic {x++;}
```

atomic

- Atomic may be annotated with outer attribute

```
__transaction [[outer]] {x++;}
```

outermost atomic

```
__transaction_relaxed {x++;}
```

relaxed

Atomic & relaxed transactions

```
__transaction_atomic {
  x++;
  if (cond)
    __transaction_cancel;
}
```

Appear to execute atomically

Can be cancelled

Unsafe statements prohibited

```
__transaction_relaxed {
  x++;
  print(x);
}
```

Cannot be cancelled

No other restrictions on content

All transactions appear to execute in serial order

Racy programs have undefined behavior

I/O Without Transactions

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
foo();
```

```
// Thread 2
foo();
```

```
Hello Concurrent Programming World!
```

```
Hello Hello Concurrent Concurrent Programming
Programming World! World!
```

...***Hello Concurrent Programming Hell World!***...
(and other fun [and appropriate] variations)

I/O With Transactions

```
void foo()
{
  cout << “Hello Concurrent Programming World!” << endl;
}
```

```
// Thread 1
__transaction_atomic
{
   foo();
}
```

```
// Thread 2
__transaction_atomic
{
   foo();
}
```

```
Hello Hello ... Hello
```

Three Hello’s?
There are only two calls?

I/O and Irrevocable Actions: Take Two

```
void foo()
{
  cout << “Hello Concurrent Programming World!” << endl;
}
```

```
// Thread 1
__transaction_relaxed
{
   foo();
}
```

```
// Thread 2
__transaction_relaxed
{
   foo();
}
```

```
Hello Concurrent Programming World!
Hello Concurrent Programming World!

(only possible answer)
```

Callable (for relaxed TX only)

- a function (including virtual functions and template functions) is intended to be called within a relaxed transaction
- intended for use by an implementation to improve the performance of relaxed transactions;
  - generate a specialized version of a transaction_callable function, and execute that version when the function is called inside a relaxed transaction.

Embedded non-transactional synchronization

Assume: x = 0, y = 0

```
__transaction_atomic {
  lock(A); ++x; unlock(A);
  lock(B); ++y; unlock(B);
}
```

Visible state:
x = 0, y = 0
x = 1, y = 1

```
__transaction_relaxed {
  lock(A); ++x; unlock(A);
  lock(B); ++y; unlock(B);
}
```

Visible state:
x = 0, y = 0
x = 1, y = 0
x = 1, y = 1

Communication via synchronization

```
__transaction_relaxed
{
    lock (L)
     sendMessage();
    unlock (L)


    lock (L)
     receiveReply();
    unlock (L)
}
```

Will deadlock for **__transaction_atomic**

```
lock (L)
 receiveMessage();
 sendReply();
unlock (L)
```

Nested ***non-transactional*** synchronization violates atomicity (isolation)

Incorrect Program

```
__transaction_atomic {
    p =new Foo();
    f(p);
}
```

data race

```
if (p != NULL)
 t = p->x;    //S
```

Racy program → undefined behavior

Practically, p **might** be NULL in S

Function Call Safety

- 3 features for safety of functions calls

1. transaction_safe attribute
2. transaction_unsafe attribute
3. Concept of implicitly declared safe function

- Different combinations offer different degrees of ability to call functions from within atomic transactions

Unsafe statements

Operations that should not nest inside atomic transactions

- Outer atomic transactions
- Relaxed transactions

Operations for which system can’t guarantee atomicity

- Access to volatile objects
- Calls to functions not declared safe
- Asm statements

Functions that break atomicity must not be declared safe

- Synchronization: operations on locks and C++0x atomics
- Certain I/O functions

Function safety attributes

Functions are declared safe via transaction_safe attribute

– May not contain unsafe statements

```
auto f = []()[[transaction_safe]] { g(); }

[[transaction_safe]] void foo() {…}
[[transaction_safe]] int (*p)();
```

Functions are explicitly declared unsafe via transaction_unsafe attribute

```
[[transaction_unsafe]] void foo() {…}
```

Template & virtual functions may have attributes

Implicit safety declarations

Non-virtual functions can be implicitly declared safe

```
void foo()  {x++;}

__transaction_atomic {
  foo();
}
```

Safe statements

Call to foo() is safe after the definition

Minimize attribute annotations

Help with template functions

Implicit safety & template functions

Safety may be unknown till instantiation

```
template <class Op>
void t(int& x, Op f) { f(x++);}
```

<u>Safe</u>

```
[[transaction_safe]] void (*p1) (int);
__transaction_atomic { t(v, p1);}
```

<u>Unsafe</u>

```
void (*p2) (int);
t(v, p2);
```

Enables reuse of template libraries

Class attributes

```
[[transaction_safe]] class C {
  void f1();
  int f2();
  [[transaction_unsafe]] void IO();
}
```

f1() & f2() are declared safe

IO() is declared unsafe

Syntactic sugar to minimize attribute annotations

Transaction expressions

Evaluate expression in a transaction

```
// exp calls copy constructor
SomeObj myObj = __transaction_atomic (exp)
SomeObj myObj = __transaction_relaxed(exp)
```

Transaction statements are not sufficient to express this pattern

A Simple Example

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

Shared access: x, y, z.

How to make safe using TM?

Option 1

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

```
Obj x, y, z;

void foo()
{
   __transaction_atomic
   {
      Obj tmp = x * y / z;

      // access tmp
   }
}
```

OK, but can cost performance (long tx).

Option 2

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

```
Obj x, y, z;

void foo()
{
   Obj tmp;

   __transaction_atomic
   {
      tmp = x * y / z;
   }

   // access tmp
```

OK, but changes behavior and suffers double assignment penalty.

Option 3

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

```
Obj x, y, z;

void foo()
{
   Obj *tmp;

   __transaction_atomic
   {
      tmp = new Obj(x * y / z);
   }

   // access tmp
```

OK, but heap (de)allocation may be slow.

Using Transaction Expressions

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   //
}
```

```
Obj x, y, z;

void foo()
{
   Obj tmp = __transaction_atomic ( x * y / z );

   // access tmp
}
```

Note: Assignment outside of tx.

Yes! This is exactly what we want.

Function transaction blocks

```
class Base {
  const int id;
  Base (int _id) :
id(_id) {}
}

class Derived : Base {
  static int count = 0;
  Derived()
__transaction_atomic
     : Base (count++) {…}
}
```

Allow to include member & base class initializers in a transaction

Challenging Example

```
class Id
{
public:
   Id(size_t id) : id_(id) {}
private:
   size_t const id_;
};

class Account : public Id
{
public:
   Account() : Id(count++) {}
private:
   static size_t count = 0;
};
```

- Challenges
  - id_ const mem
  - must be init'd in mem initialization
  - count is static (shared memory)
  - synchronize access to count or racy program
- Many STM ***cannot*** handle this

C++ TM Spec *Can* Handle This

```
class Id
{
public:
   Id(size_t id) : id_(id) {}
private:
   size_t const id_;
};

class Account : public Id
{
public:
   // member initialization atomic / isolated
   Account() __transaction_atomic : Id(count++) {
... }
private:
   static size_t count = 0;
};
```

When I first saw this, the only word that came to mind was

“Wow!”

Optimizing Atomicity

```
class Object
{
public:
   // initialization atomic/isolated
   Object() __transaction_atomic :
      arr_(alloc_.allocate()) { ... }
```

Try doing this with std::mutex

Disclaimer: it can be done.

Challenging to write correctly and efficiently!

TM doesn't unnecessarily limit parallelism.

Cancel and Cancel-throw forms (may be removed in future)

- 2 forms of cancel
  1. A basic cancel statement to cancel immediate enclosing atomic transaction
  2. cancel [[outer]] statement that cancels the enclosing outer atomic transaction
- 2 forms of cancel-throw statements
  1. A basic cancel and throw statement
  2. cancel and throw [[outer]]

Cancel Statement

```
__transaction {
  ...
  __transaction_atomic {
    x++;
    if (cond)
     __transaction_cancel;
  }

y++;
}
```

- Rolls back innermost atomic transaction statement
- Continues with the following statement
- Must be in lexical scope of atomic statement
- Cannot be applied to transaction function blocks & relaxed transactions

Cancel & unsafe actions

```
__transaction_relaxed {
  bool all_ok = false;
  __transaction_atomic {
    ...
    if (all_is_ok())
      all_ok = true;
    else
      __transaction_cancel;
  }
  if (all_ok) IO();
}
```

This demonstrates Cancel and unsafe stmts cannot execute in the same transaction

But this can be done via combined atomic nested in relaxed

Demonstrates use of both types

Cancel outer statement

```
__transaction_atomic [[ outer ]] {
  y++;
  __transaction_atomic {
    x++;
    if (cond)
     __transaction_cancel [[ outer ]];
  }
}
```

Rolls back outer atomic transaction
Must be in dynamic scope of outer transaction
Needs transaction_may_cancel_outer function attribute

May cancel outer attribute

Specifies that a function may contain cancel outer statement in its dynamic scope

```
[[transaction_may_cancel_outer]] void f1(){
  ... __transaction_cancel [[outer]];
}
[[transaction_may_cancel_outer]] void f2(){
  f1();
}
__transaction_atomic [[outer]] { f2();}
```

Declares a function safe; can be used on function pointers

What happens on an exception?

```
__transaction_atomic {
  x++;
  if (cond)
    throw 1;
}
```

When integer escapes the transaction

- Should the effects of x++ be committed?
- Or should they be rolled back?

Active debate in community

Both sides are right

Some programs behave surprisingly under commit-on-escape

Others under rollback-on-escape

Observations:

- Exceptions that can unexpectedly escape a transaction are potentially dangerous
- No single behavior appropriate for all cases
  - Only the programmer can determine what's appropriate

Our approach

Support both semantics & let programmer decide

New syntax for

- Exception specifications on transaction statements, or expressions, but not blocks
- Throwing exceptions that roll back a transaction
- Allowed on atomic and relaxed

Exception specification (may change in future)

Specify whether an exception is allowed to propagate outside of scope (xxx=atomic/relaxed)

```
__transaction_xxx noexcept(true)  {…}// not allowed
__transaction_xxx noexcept        {…}// not allowed
__transaction_xxx noexcept(false) {…}// allowed
```

Terminate if contract violated

No specification?

Terminate if exception does not match

```
__transaction_xxx                 {…}// default allowed
```

Default: **all** exceptions allowed to escape

- Consistent with C++11 exception specifications
- Use at your own risk

Commit-on-exception

Standard syntax for exception throw

```
__transaction_atomic noecept(true) {
  try {
     throw 1;
  } catch ( int & e) {
      …; //exception caught here
  }
}
```

Easy to specify that any exception may commit

```
__transaction_atomic noexcept {
  exception_throwing_fun();
}
```

Rollback-on-exception

Syntax for exception throw

```
try {
    __transaction_atomic noexcept(false){
    try { ...
        __transaction_cancel throw 1;
    } catch (int& e) {
        assert(0); //never reached!
    }
} catch (int &e) {
    cout <<“Caught e!” << endl;
}
```

Exception must be enums or integrals

Restrict exceptions to enum / integral?

```
try
{
   __transaction_atomic noexcept(false)
   {
      ...
      __transaction_cancel
         throw TxException(txState);
   }
}
catch (TxException &e)
{
   cout << e.state();  // CRASH!
}
```

Accessing state that no longer exists.

Agenda

- STM, HTM, HybridTM
- Birth of a specification
- Design Goals
- Motivation for SG5 in C++ Standard
  - Use cases
  - Usability
  - Performance
- Language Constructs
  - Transactions, atomic and relaxed
  - Race-free semantics
  - Unsafe statements
  - Attributes
  - Transaction expressions and try blocks
  - Cancel
  - Exception handling
- **SG5 Progress**

Straw poll results from Bristol

- Change the term "relaxed transaction“ : SF: 3, F:2, N:2, WA: 2, SA: 2.
- Straw poll: explicit cancellation: SF:0, WF: 4, N:0, WA: 8, SA: 1
- Straw poll: ability to cancel on escaping exception: SF:4, WF: 2, N:1, WA: 6, SA: 0
- Straw poll: provide syntax support for commit on escaping exception: SF:4, WF:4, N:0, WA:2, SA:2
- Straw poll: at least enough support for simple data escape (integral and enumeration types only): SF:3, WF:3, N:2, WA:1, SA:2
- Straw poll: advanced data escape along Torvald's presentation: SF:0, WF:0, N:2, WA:3, SA:6
- Straw poll: atomic transactions: SF:6, WF:1, N:2, WA:2, SA:0
- Straw poll: relaxed transactions: SF:5, WF:3, N:1, WA:2, SA:0

Latest current status

- Intend to file for a TS NP/CD in October 2013, Chicago Standard meeting
- Presented at ACCU2013, ADC++2013, C++NOW 2013
- Many people very interested in supporting work
- Started work on standarese wording
  - Object model
  - Syntax support
  - Library safety

Summary

TM adoption requires common high-level interfaces

- SG5 Opens path for standard language extensions & semantics

  -proposed draft TS specification for 2017

  –hardware is here and now

  - would you want C++ 2017, or 2022 that has no TM support?

We need feedback, all are welcome to join:

https://groups.google.com/forum/?hl=en&fromgroups=#!forum/c-tm-language-

My blogs and email address

- **http://isocpp.org/wiki/faq/wg21:michael-wong**
  OpenMP CEO: http://openmp.org/wp/about-openmp/
  My Blogs: http://ibm.co/pCvPHR
  C++11 status: http://tinyurl.com/43y8xgf
  Boost test results
  http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27006911
  C/C++ Compilers Support/Feature Request Page
  http://www.ibm.com/developerworks/rfe/?PROD_ID=700
  TM: https://sites.google.com/site/tmforcplusplus/
- Chair of WG21 SG5 Transactional Memory
- IBM and Canada C++ Standard Head of Delegation
- ISOCPP.org Director, Vice President
- Tell us how you use OpenMP:
- http://openmp.org/wp/whos-using-openmp/