Transactional Language Constructs for C++

vs.

2012-05-08
Justin Gottschlich on behalf of the C++ TM Drafting Group

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

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 do 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 straightforward 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:

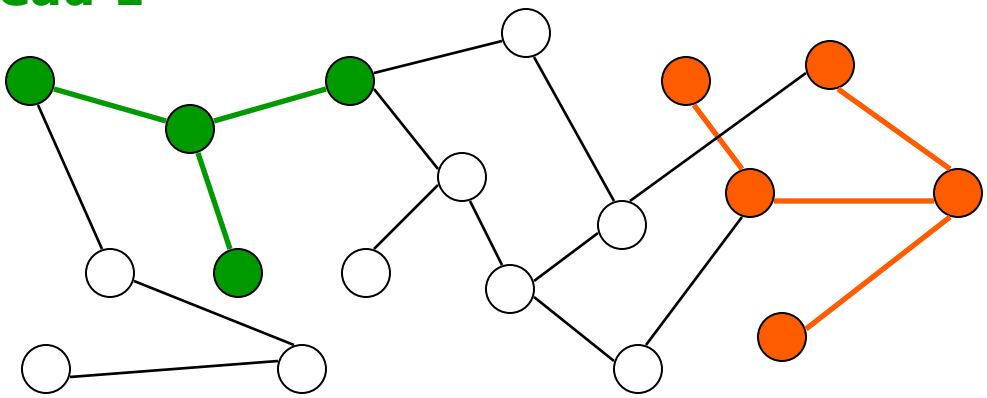
```
class ImpT
{
  ImpT& operator=(ImpT T& rhs)
  {
    transaction {
      // handle assignment
    }
  }
};
```

Impossible to deadlock.

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



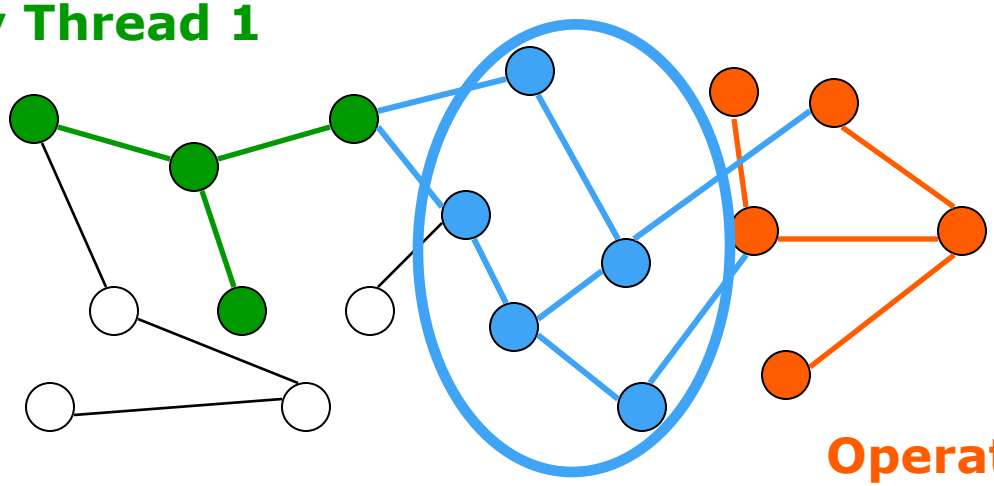
Operation by Thread 2

Why Not Locks?

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

How do you implement this?
Operations by both Thread 1 and 2

Operation by Thread 1



Operation by Thread 2

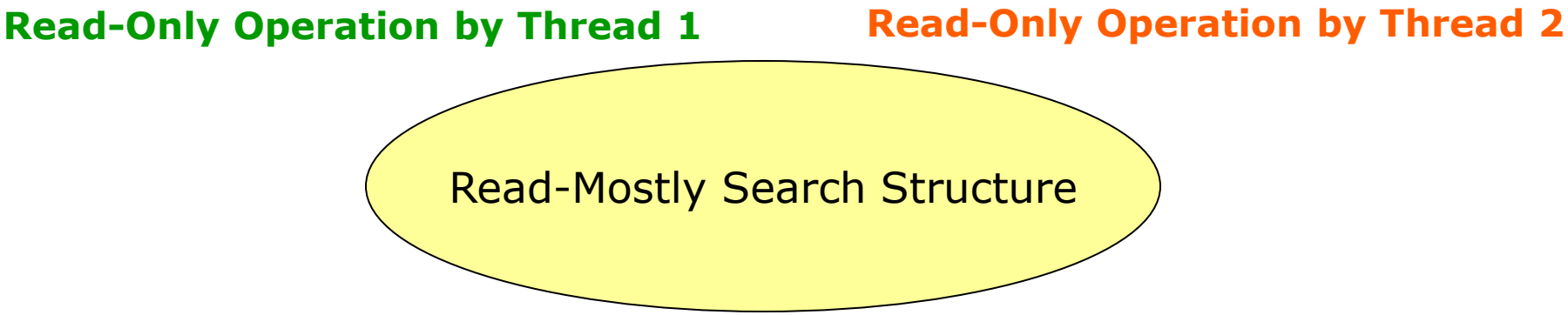
Composition / Modularity *(Herb's Opening Comments)*

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

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

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)

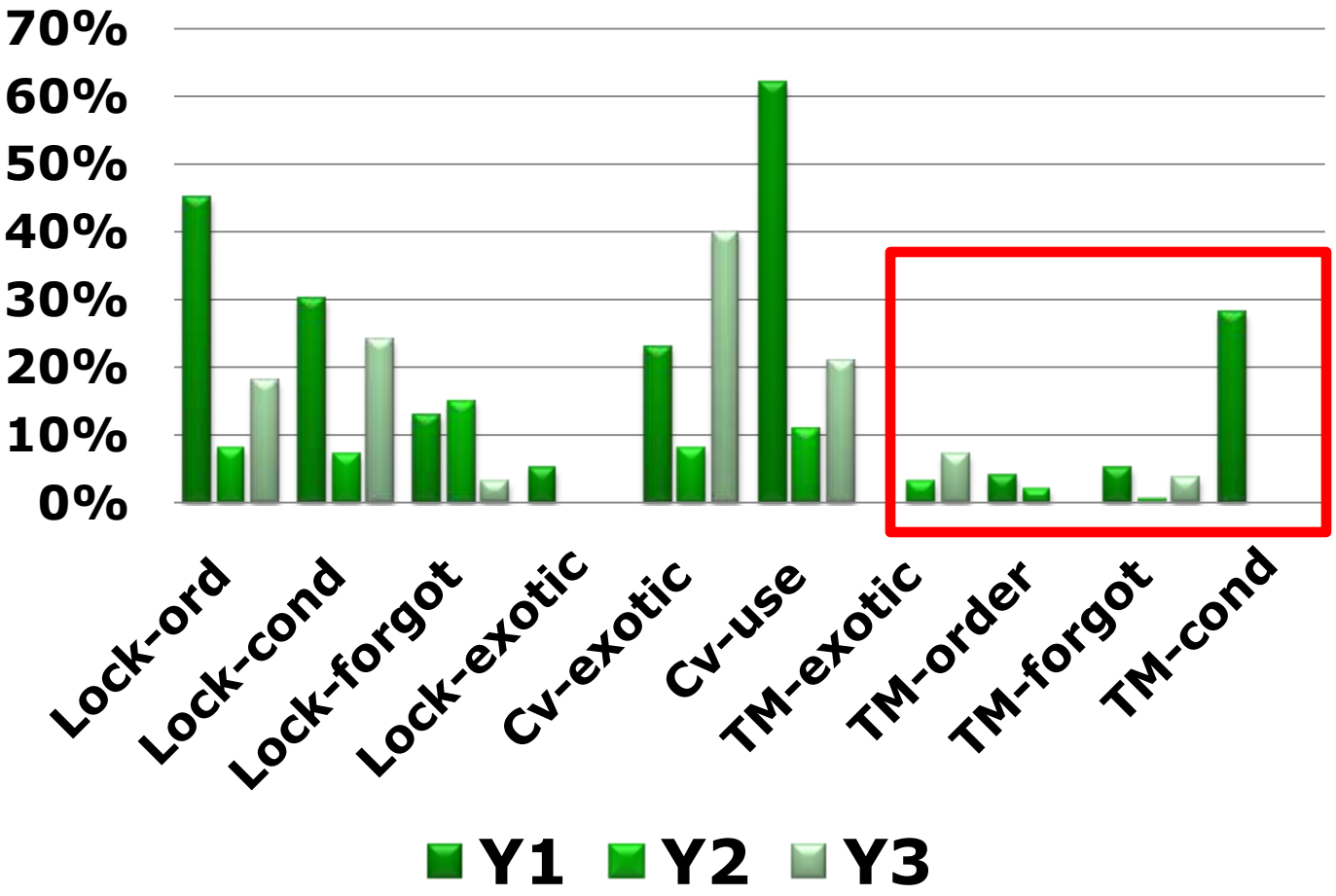


Usability

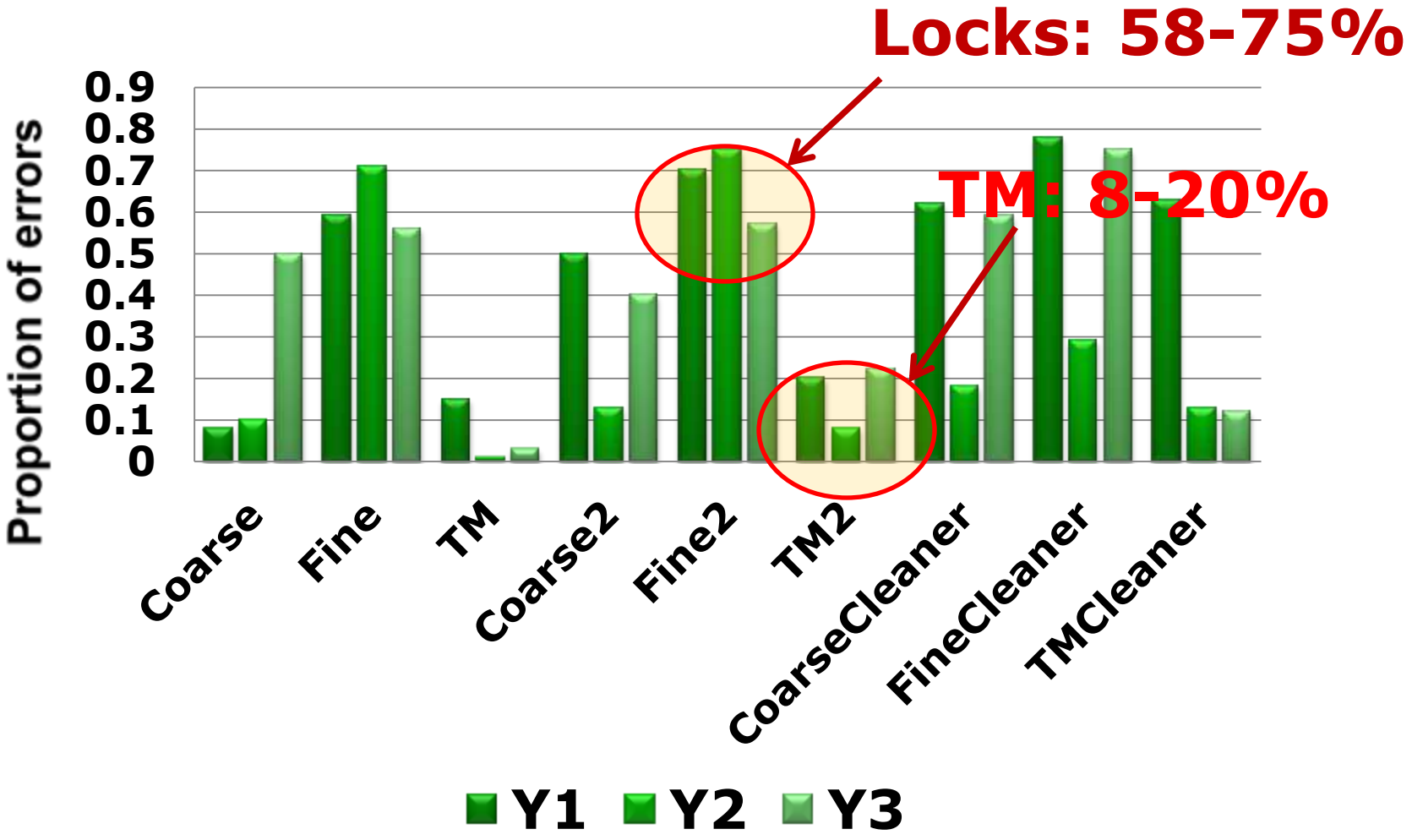
Two User Studies

- 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

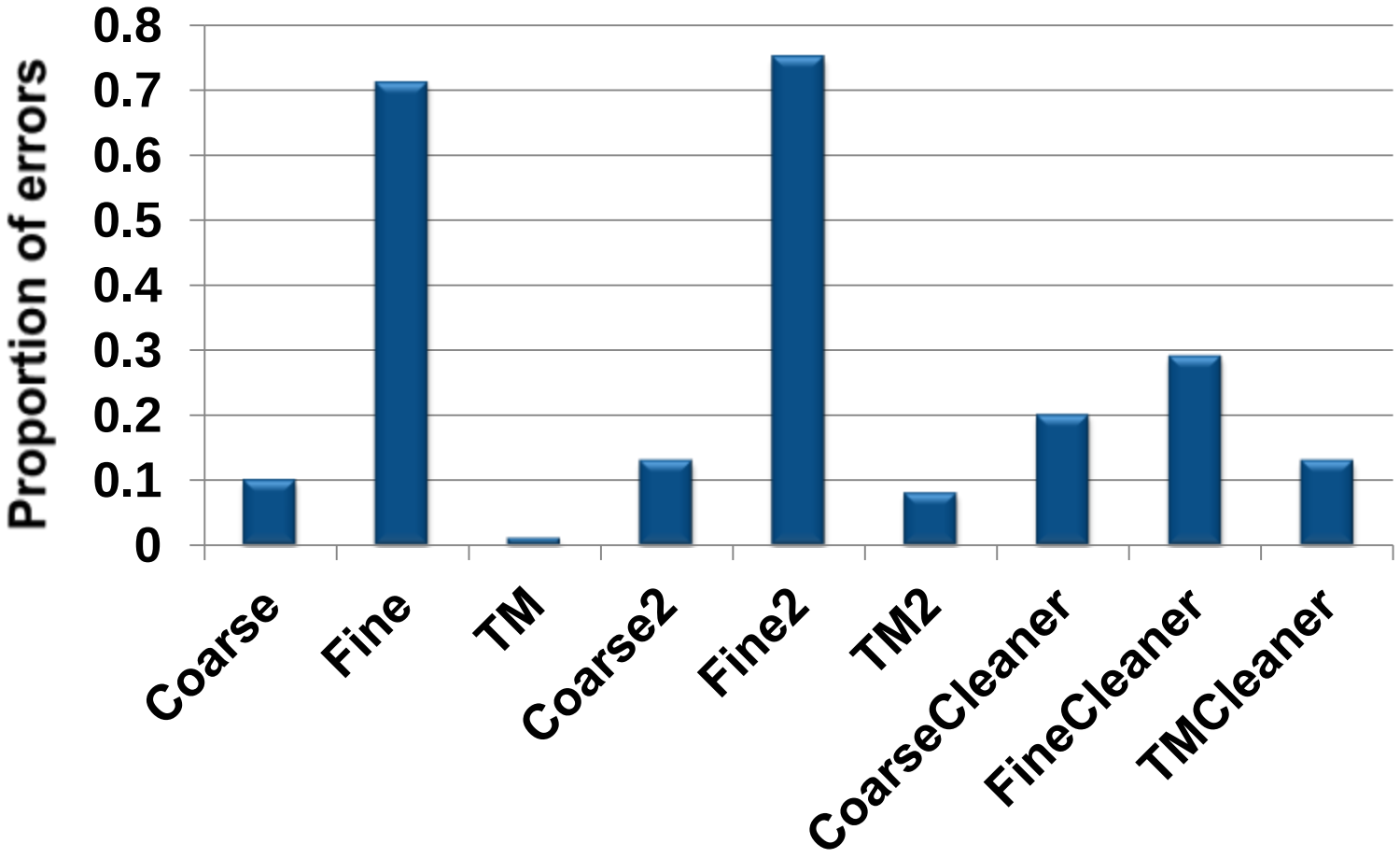
Error Rates by Defect Type



Overall Error Rates



Overall Error Rates: Year 2



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

Code

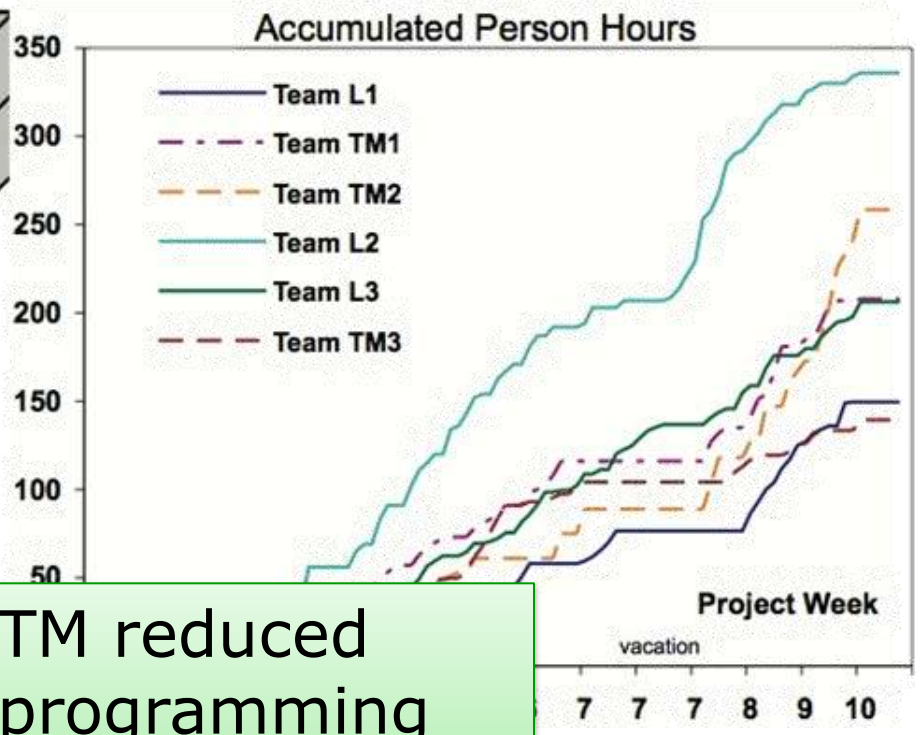
- Average LOC about the same
- TM teams have fewer LOC with parallel constructs (2%-5% vs. 5%-11%)

	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**		

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*	*14*	*2*	83

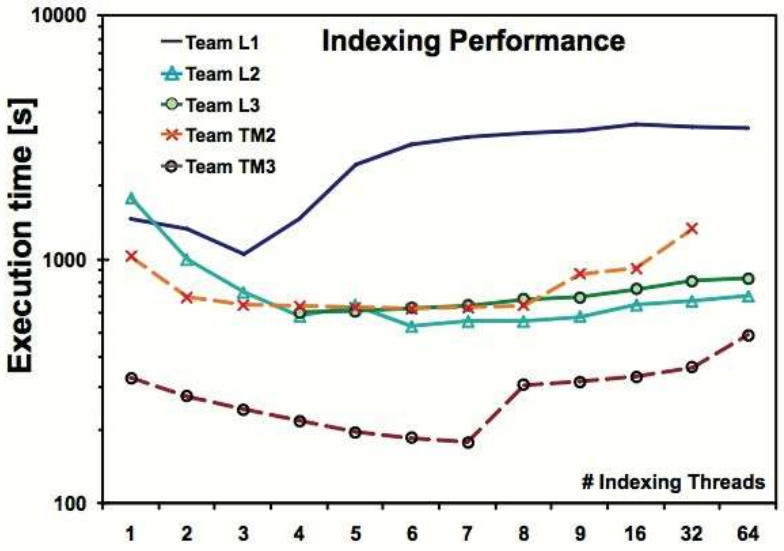


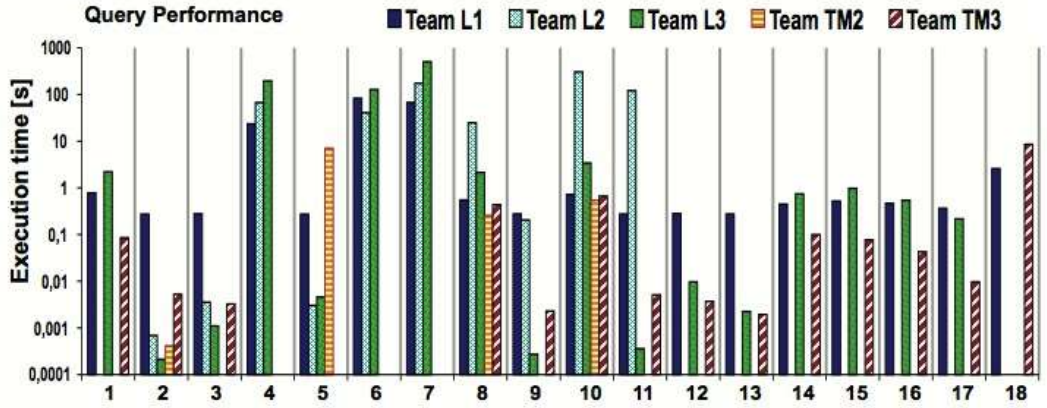
TM reduced programming effort by ~14%

Less for TM

teams in last weeks: Refactoring transactions, performance problems, experiments

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
- Demonstration that TM performance need not be bad in practice

Performance

Is TM Fast Enough?

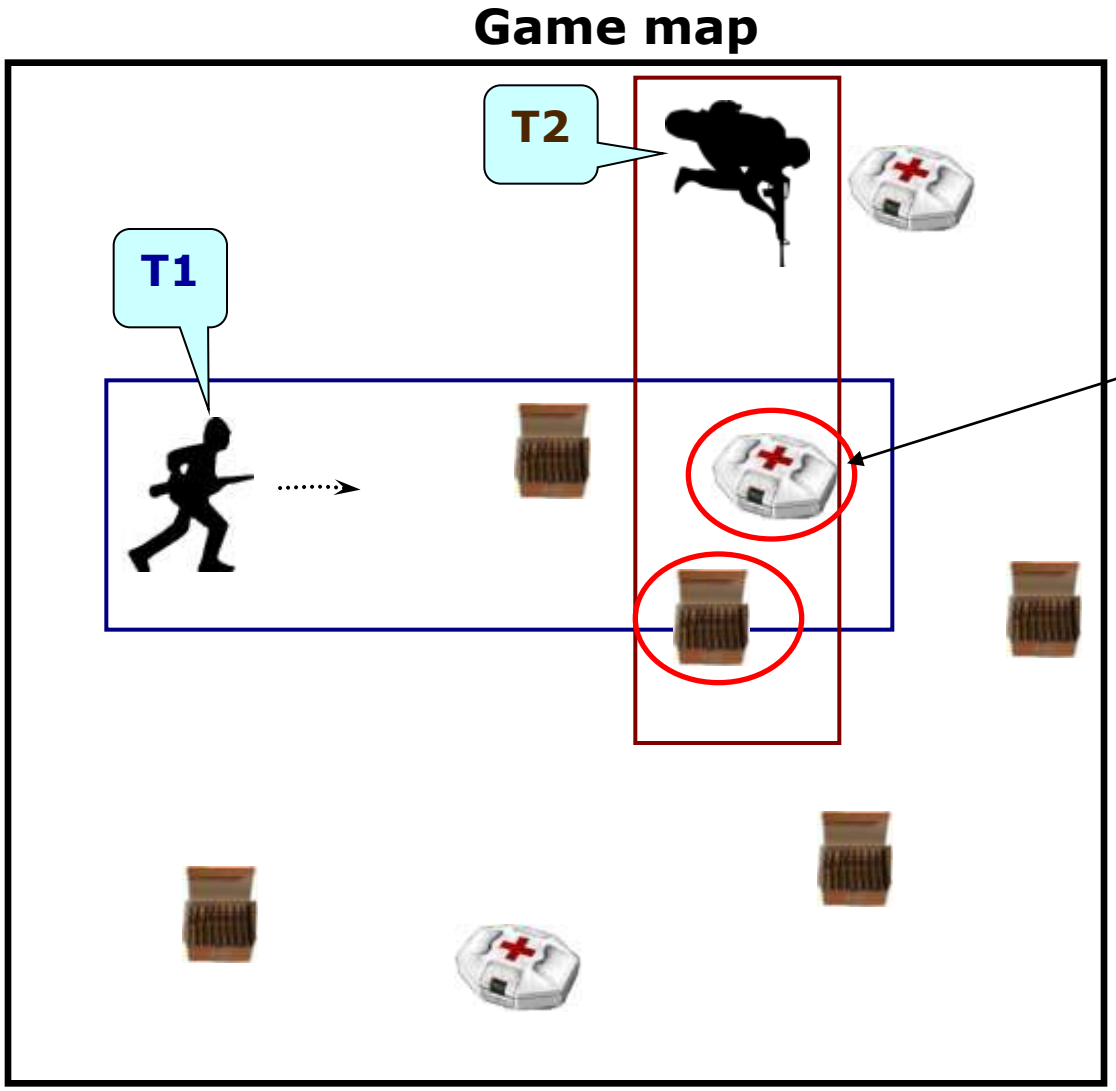
- Many different STMs with different goals (and different guarantees)
  - TL2: baseline state-of-the-art
  - TinySTM: added safety guarantees (opacity)
  - NOrec: generalized support of many features
  - InvalSTM: contention-heavy programs
  - SkySTM: scalable to upwards of 250 threads
- How to choose?
  - Use adaptive algorithm (Wang et al., HiPEAC'12)
  - ***Change TM without changing client code***

Multiplayer games

- More than 100k concurrent players
- "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
  - SynQuake, simulates Quake battles
  - Software-only TM (STM)
  - Presented at EuroSys 2010

Game server is the bottleneck

Conflicting player actions



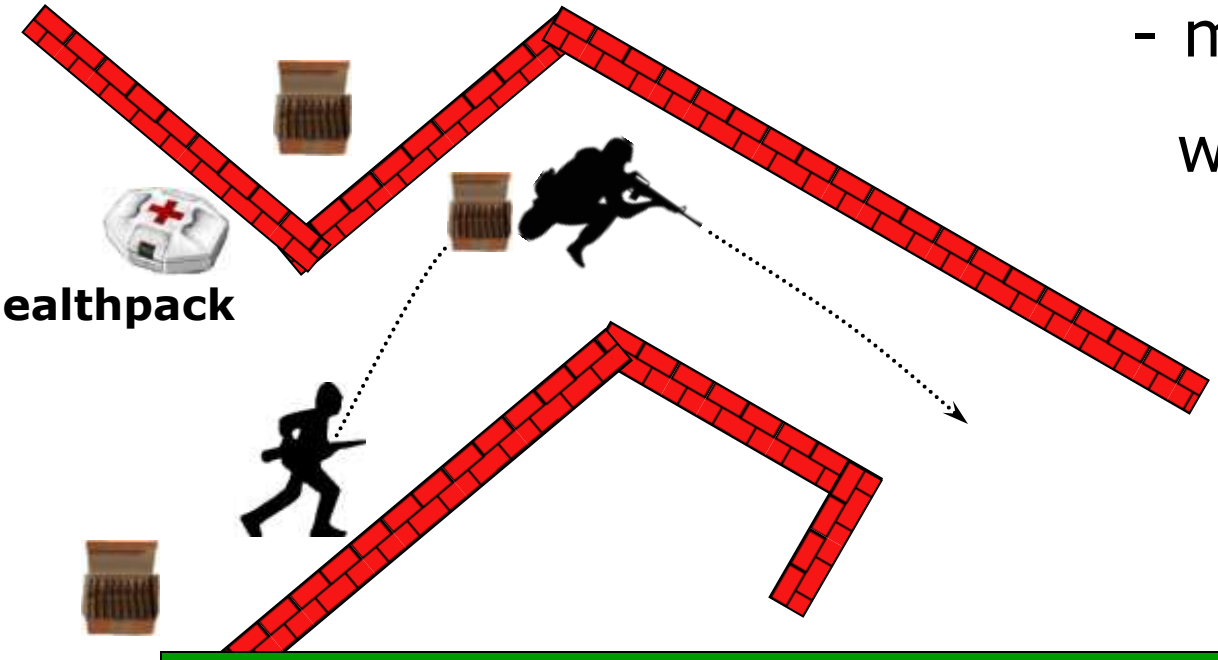
Need for synchronization

Player actions

Compound action:

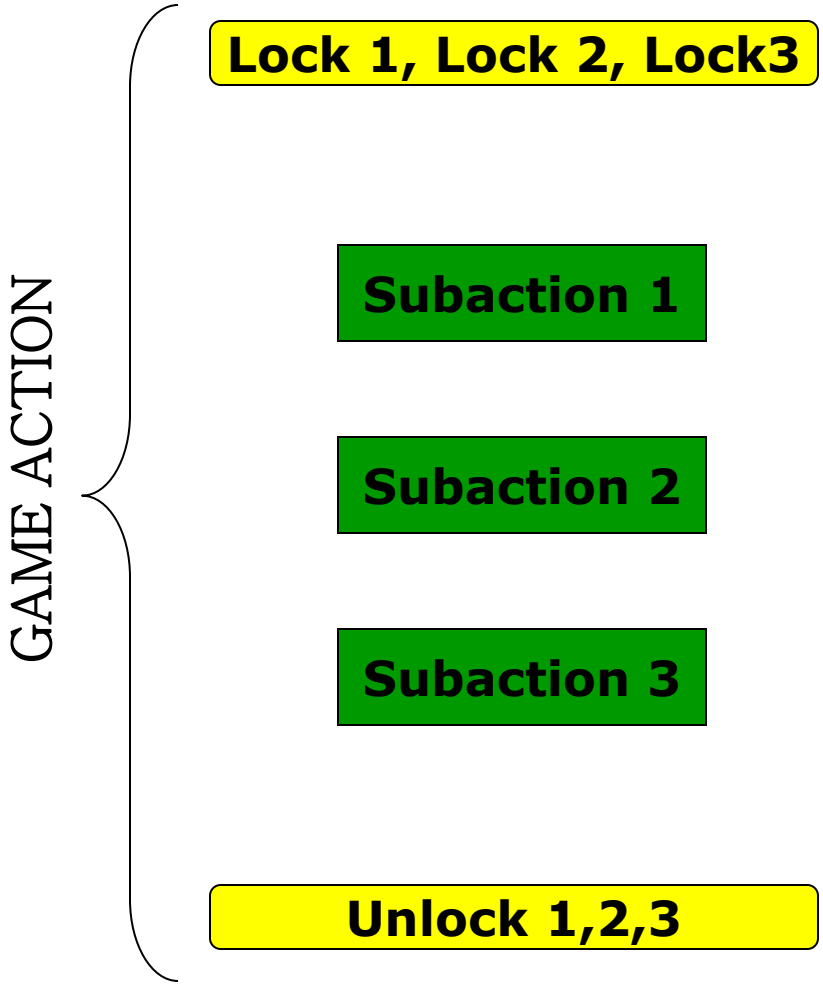
- move, charge

  weapon and shoot



Requirement:
consistency and atomicity
of whole game action

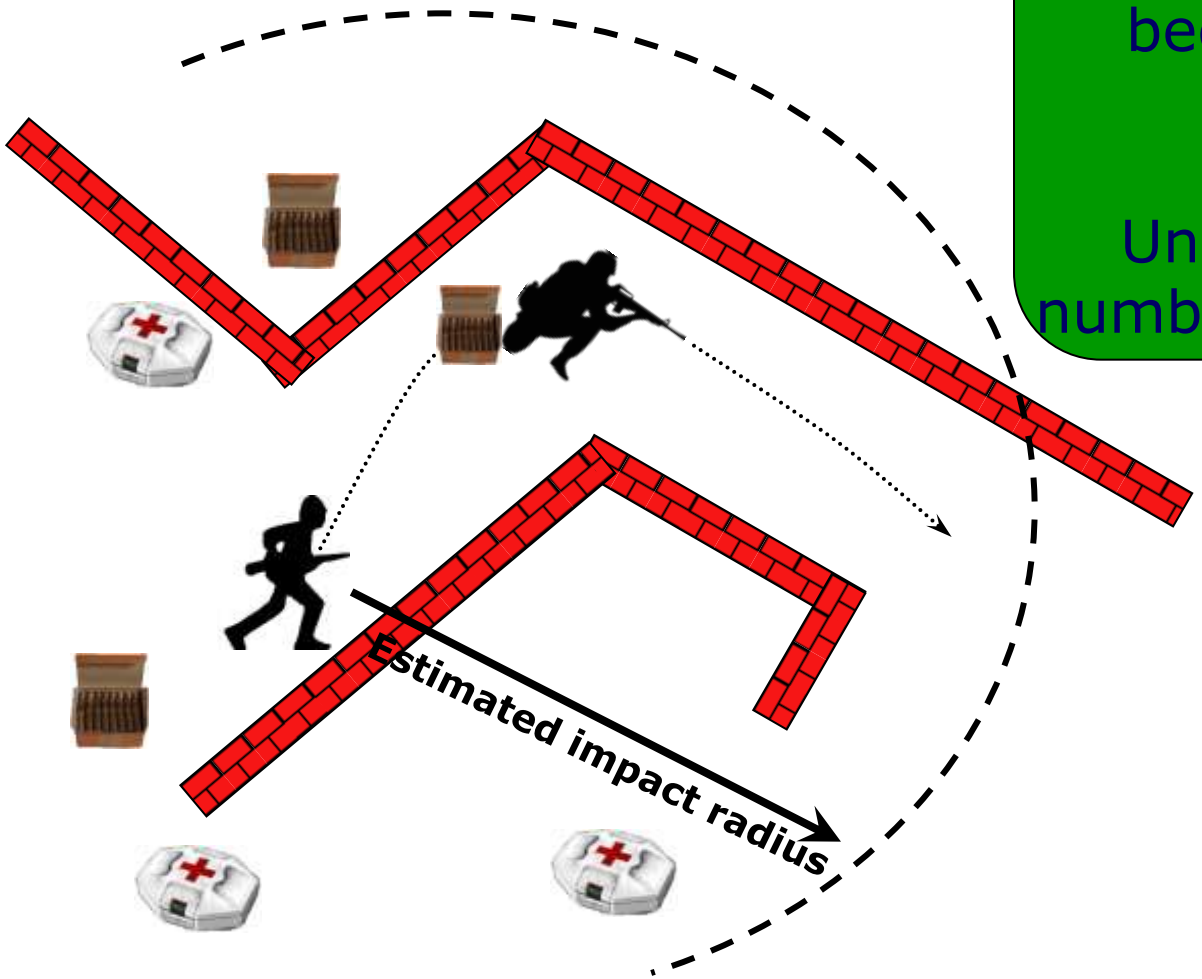
Conservative locking



Conservatively acquire all locks at beginning of action

Problem 1:
Unnecessarily long conflict duration

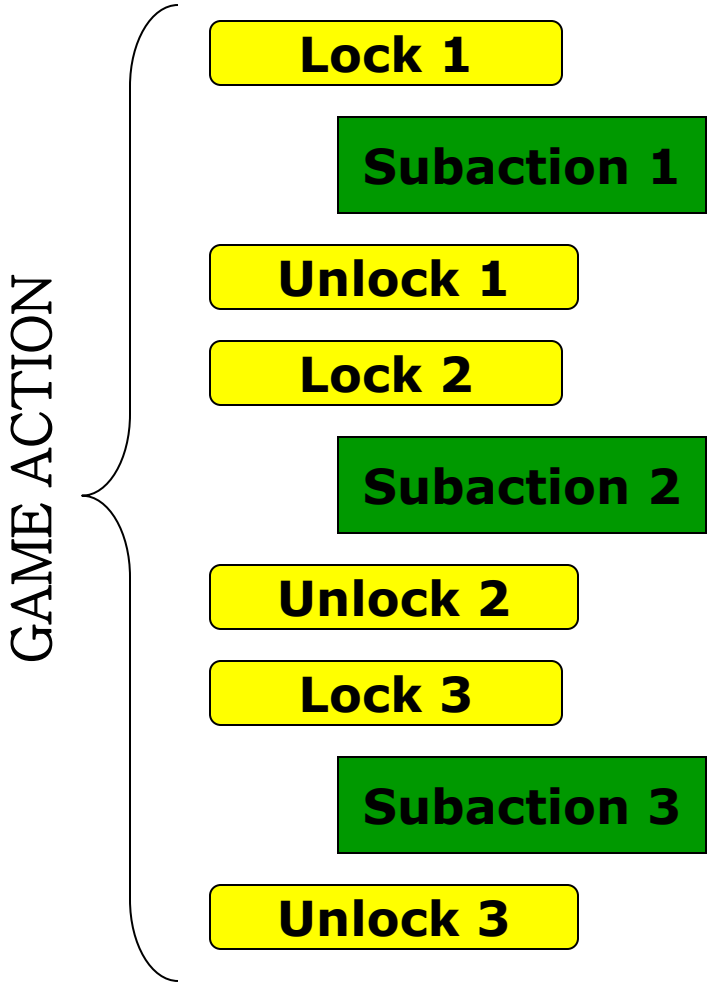
Conservative locking



Conservative estimate of impact range at beginning of action

Problem 2:
Unnecessarily high number of locked objects

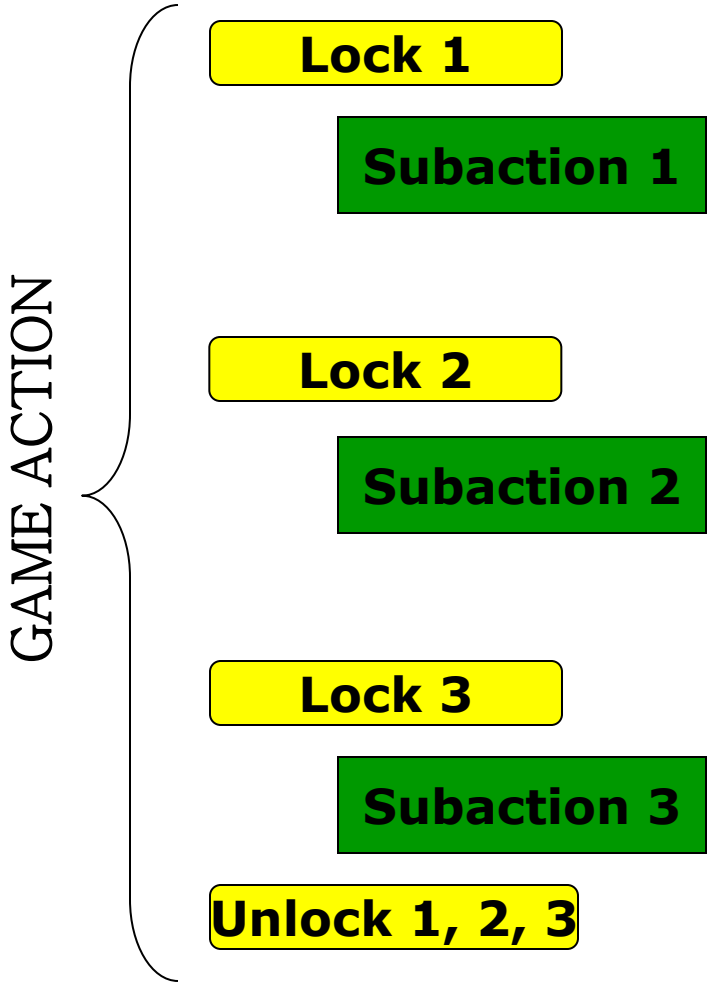
Fine-grained locking?



Not possible !

Problem:
- No atomicity for whole action

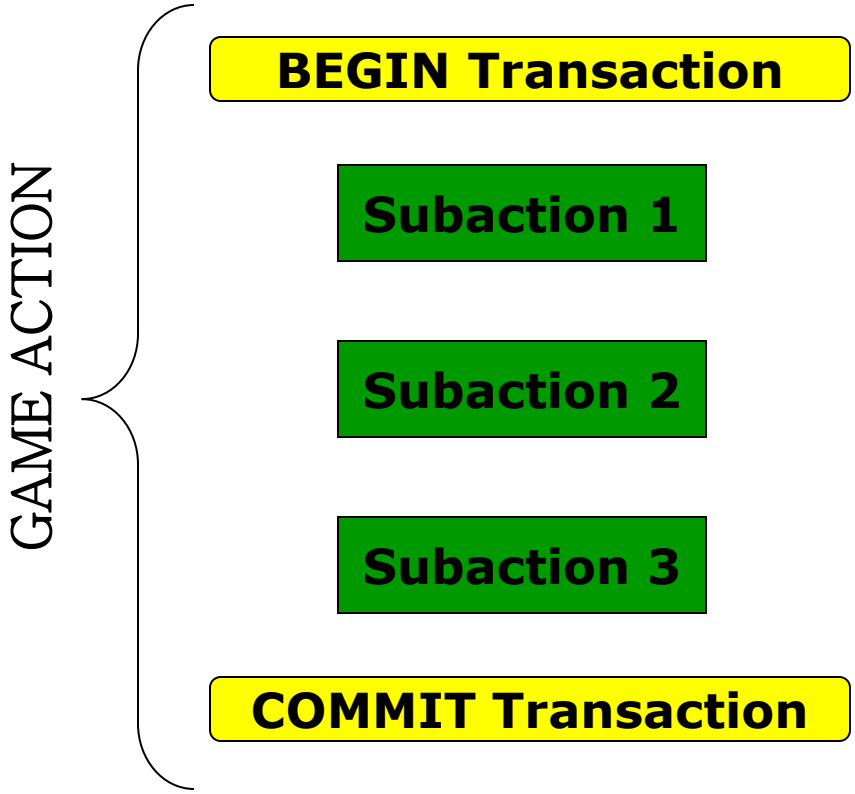
Fine-grained locking?



Not possible !

Problem:
- Deadlocks

STM - Synchronization

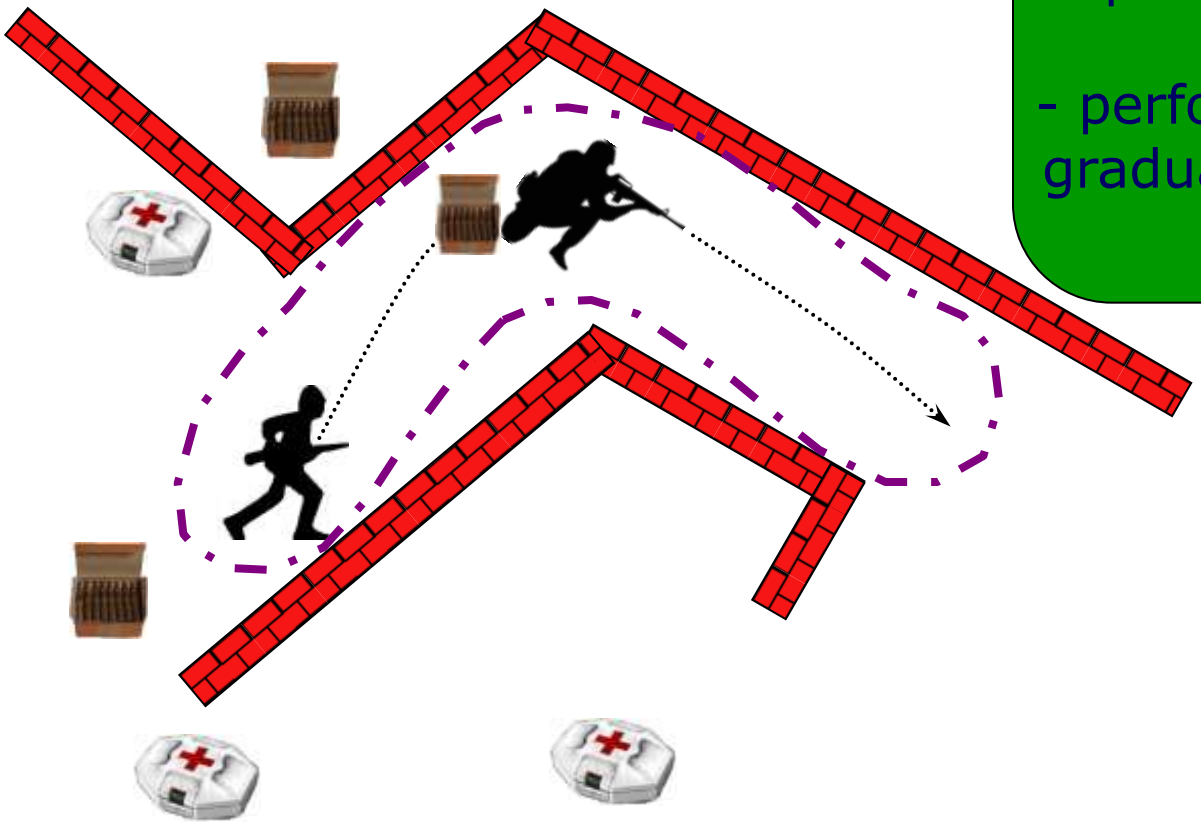


Problems solved:

- Deadlocks
- Atomicity

Handled automatically

STM - Synchronization



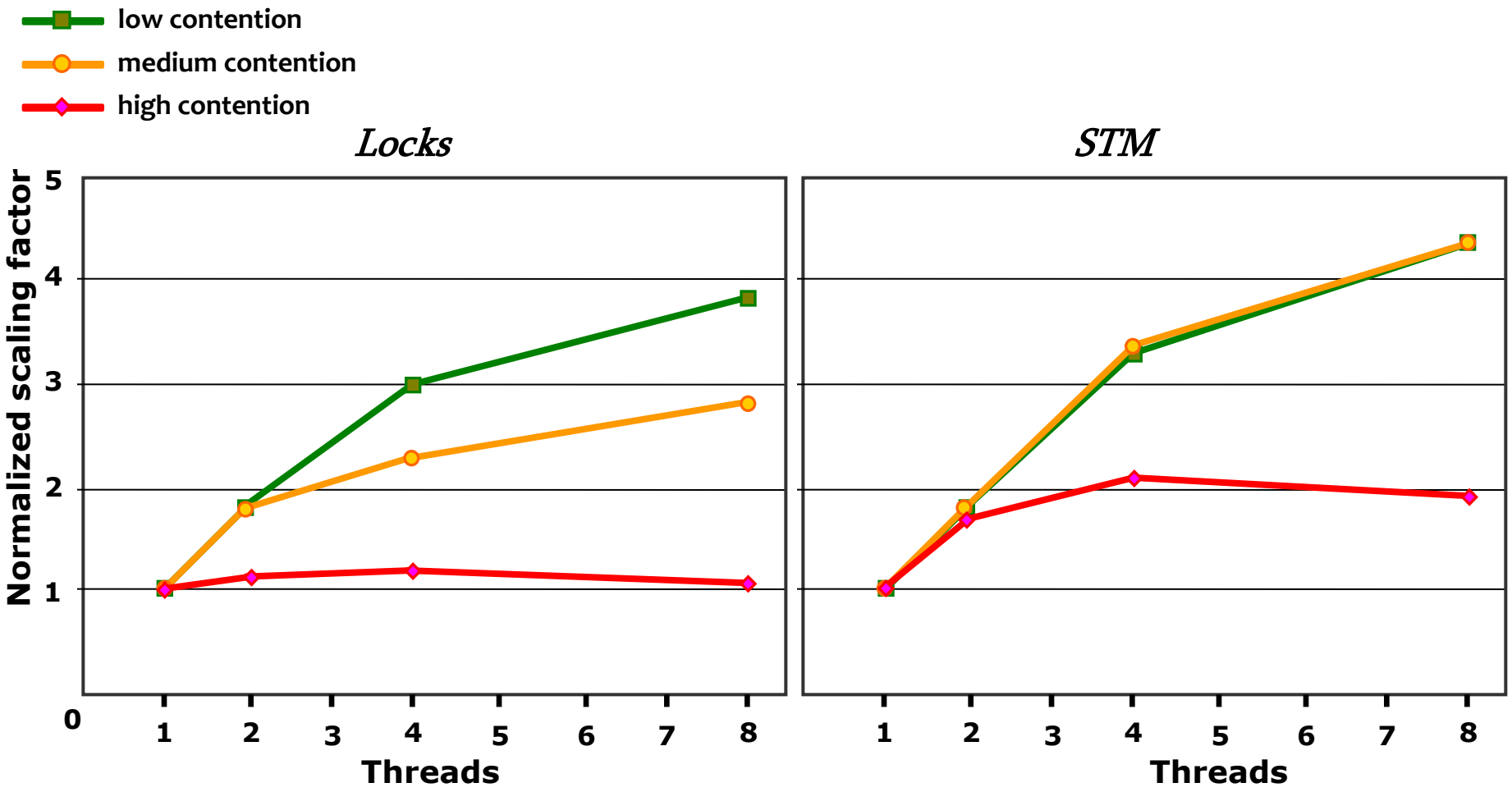
Collision detection optimized:

- split action into subactions

- perform collision detection gradually for each subaction

Scalability

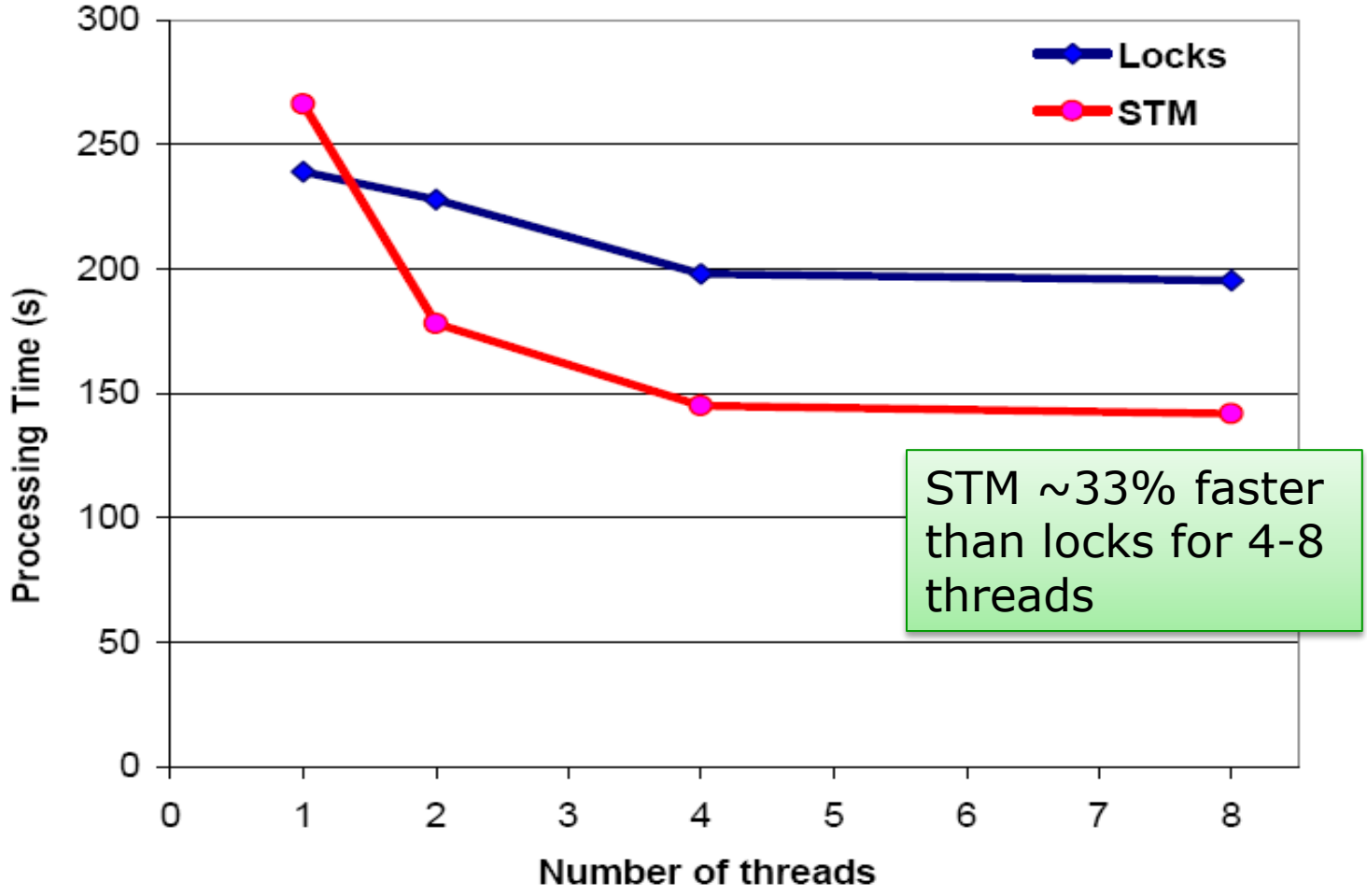
8 core machine



STM scales better in all 3 contention scenarios

Processing Times

Medium contention



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

Thank you! Questions?

Justin Gottschlich
justin.e.gottschlich@intel.com