Is Transactional Programming Actually Easier?

Christopher J. Rossbach,
Owen S. Hofmann,
Emmett Witchel
University of Texas at Austin, USA

Transactional Memory: Motivation Mantra

- We need better parallel programming tools
  - (Concurrent programming == programming w/locks)
  - Locks are difficult
  - CMP ubiquity → urgency
- Transactional memory is “promising”:
  - No deadlock, livelock, etc.
  - Optimistic → likely more scalable
- **Conclusion**:
  - Transactional Memory is ***easier*** than locks
  - Corollary: All TM papers should be published

Is TM *really* easier than locks?

- Programmers ***still must write critical sections***
- Realizable TM will have *new* issues
  - HTM overflow
  - STM performance
  - Trading one set of difficult issues for another?
- Ease-of-use is a critical motivator for TM research

It's important to know the answer to this question

How can we answer this question?

Step 1: Get some programmers

(preferrab

Step 2: hav

program

Step 3: Ask

Step 4: Eva

This talk:

- TM vs. locks user study
- UT Austin OS undergrads
- same program using
  - locks (fine/coarse)
  - monitors
  - transactional memory

Outline

- Motivation
- Programming Problem
- User Study Methodology
- Results
- Conclusion

The programming problem

sync-gallery: a rogue's gallery of synchronization

- Metaphor → shooting gallery (welcome to Texas)
- Rogues → shoot paint-balls in lanes (1 red, 1 blue)
- Cleaners → change targets ba

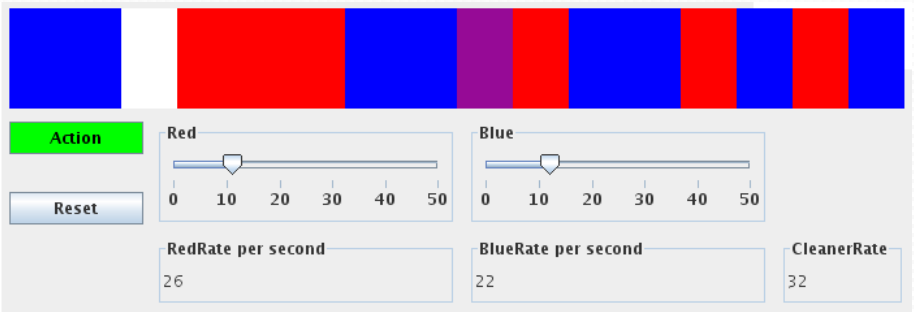
Sync-gallery invariants

- Only one shooter per lane **(Uh, hello, dangerous?!)**
- Don't shoot colored lanes **(no fun)**
- Clean only when all lanes shot **(be lazy)**
- Only one cleaner at a time

Shot by both rogues

Sync-gallery Implementations

- **Program specification variations**
  - Single-lane
  - Two-lane
  - Cleaner (condition vars + additional thread)
- **Synchronization primitive variations**
  - Coarse: single global lock
  - Fine: per lane locks
  - Transactional Memory



Variation 1: “single-lane rogue”

```
Rogue() {
  while(true) {
    Lane lane = randomLane();
    if(lane.getColor() == WHITE)
      lane.shoot();
    if(allLanesShot())
      clean();
  }
}
```

globalLock.lock()

beginTransaction()

lane.lock()

lane.unlock()

lockAllLanes() ???

endTransaction()

globalLock.unlock()

Coarse-grain locking

Fine-grain locking

Transactions

Invariants:

- One shooter per lane
- Don’t shoot colored lanes
- One cleaner thread
- Clean only when all lanes shot

Variation 2: "two-lane rogue"

```
Rogue() {
  while(true) {
    Lane a = randomLane();
    Lane b = randomLane();
    if(a.getColor() == WHITE &&
       b.getColor() == WHITE) {
      a.shoot();
      b.shoot();
    }
    if(allLanesShot())
      clean();
}}
```

globalLock.lock()

a.lock();
b.lock();

Requires lock-ordering!

lockAllLanes() ???

globalLock.unlock()

Invariants:

- One shooter per lane
- Don't shoot colored lanes
- One cleaner thread
- Clean only when all lanes shot

Coarse-grain locking / **Fine-grain locking**

Variation 3: "cleaner rogues"

```
Rogue() {
  while(true)
    Lane lane = randomLane();
    if(lane.getColor() == WHITE)  if(allLanesShot())
      lane.shoot();                 lanesFull.signal();
} }
Cleaner() {                      while(!allLanesShot()
  while(true) {                    lanesFull.await()
    if(allLanesShot())
      clean();
} }
```

(still need other locks!)

Invariants:

- One shooter per lane
- Don't shoot colored lanes
- One cleaner thread
- Clean only when all lanes shot

Synchronization Cross-product

	Coarse	Fine	TM
Single-lane	Coarse	Fine	TM
Two-lane	Coarse2	Fine2	TM2
Cleaner	CoarseCleaner	FineCleaner	TMCleaner

9 different Rogue implementations

Outline

- Motivation
- Programming Problem
- User Study Methodology
  - TM Support
  - Survey details
- Results
- Conclusion

TM Support

- Year 1: DSTM2 **[Herlihy 06]**
- Year 2+3: JDASTM **[Ramadan 09]**
- Library, not language support
  - No atomic blocks
  - Read/write barriers encapsulated in lib calls
  - Different concrete syntax matters

DSTM2 concrete syntax

```
Callable c = new Callable<Void> {
   public Void call() {
      GalleryLane l = randomLane();
      if(l.color() == WHITE))
         l.shoot(myColor);
      return null;
   }
}
Thread.doIt(c); // ← transaction here
```

JDASTM concrete syntax

```
Transaction tx = new Transaction(id);
boolean done = false;
while(!done) {
   try {
      tx.BeginTransaction();
      GalleryLane l = randomLane();
      if(l.TM_color() == WHITE))
         l.TM_shoot(myColor);
      done = tx.CommitTransaction();
   } catch(AbortException e) {
      tx.AbortTransaction();
      done = false;
   }}
```

Undergrads: the ideal TM user-base

- TM added to undergrad OS curriculum
- Survey accompanies sync-gallery project
- Analyze programming mistakes
- TM's benchmark for success
  - ***Easier to use than fine grain locks or conditions***

Survey

- Measure previous exposure
  - Used locks/TM before, etc
- Track design/code/debug time
- Rank primitives according along several axes:
  - Ease of reasoning about
  - Ease of coding/debugging
  - Ease of understanding others' code

http://www.cs.utexas.edu/~witchel/tx/sync-gallery-survey.html

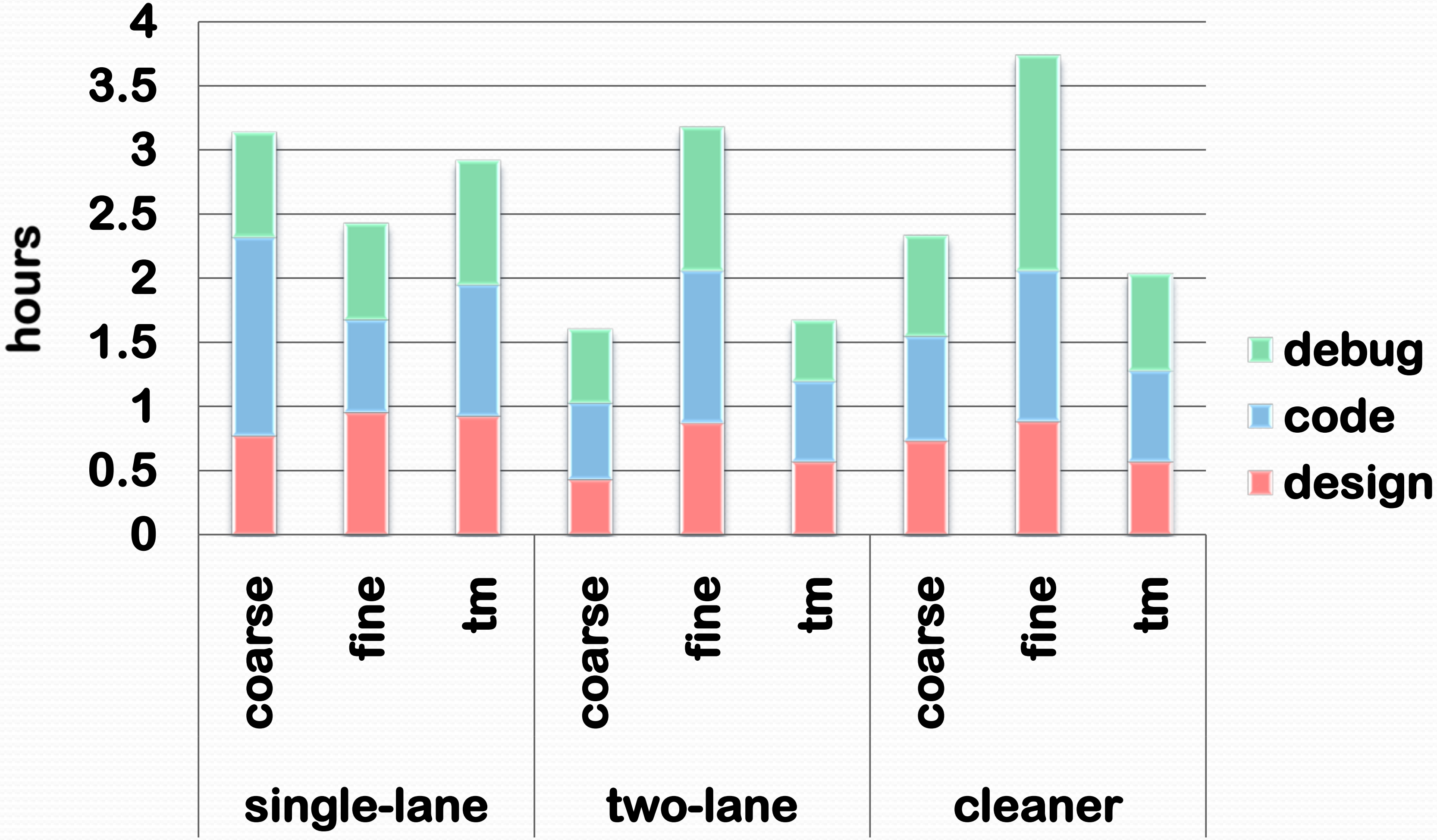
Data collection

- Surveyed 5 sections of OS students
  - 2 sections x 2 semesters + 1 section x 1 semester
  - **237** students
  - **1323** rogue implementations
- Defect Analysis
  - Automated testing using condor
  - Examined all implementations

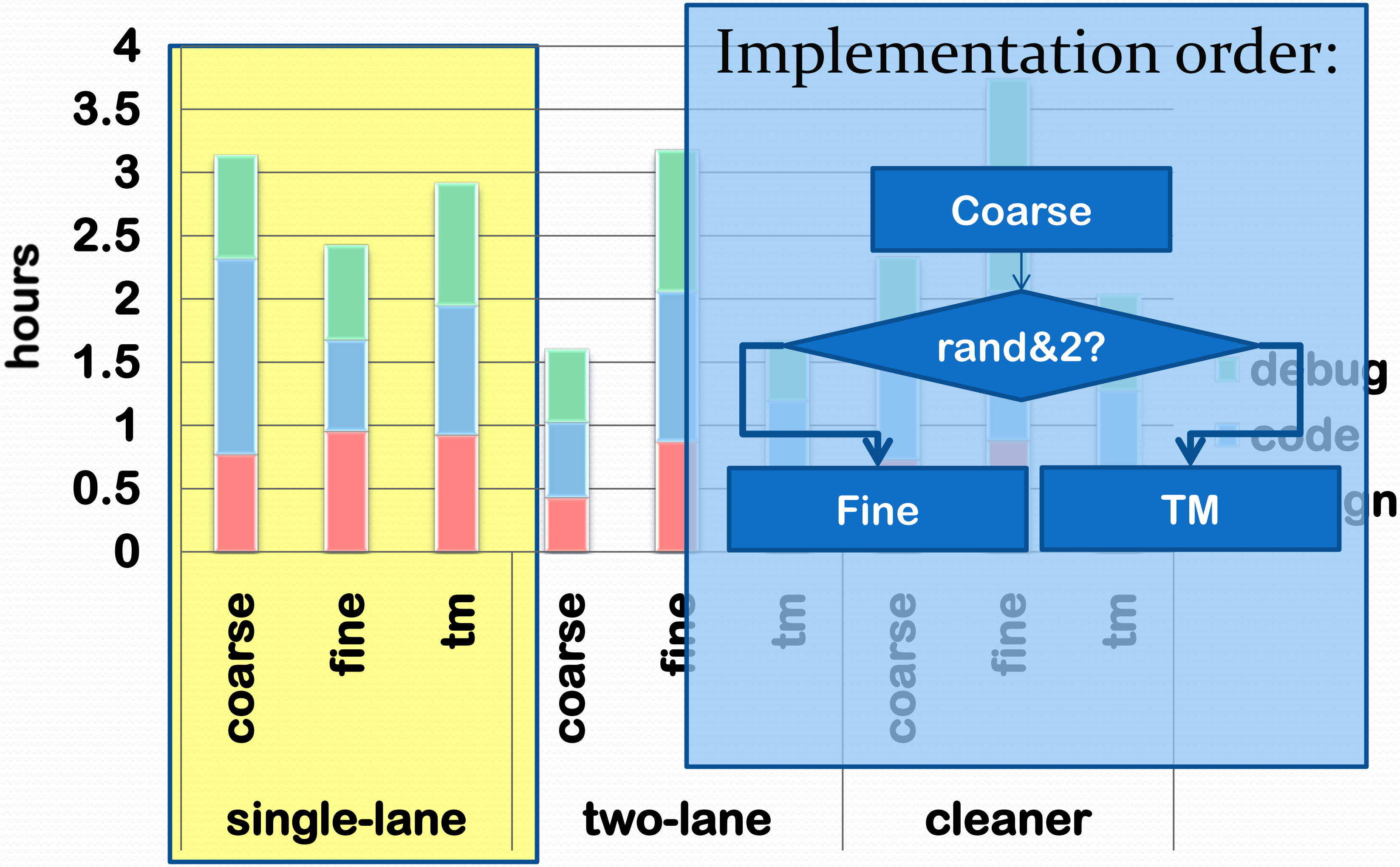
Outline

- Motivation
- Programming Problem
- User Study Methodology
- Results
- Conclusion

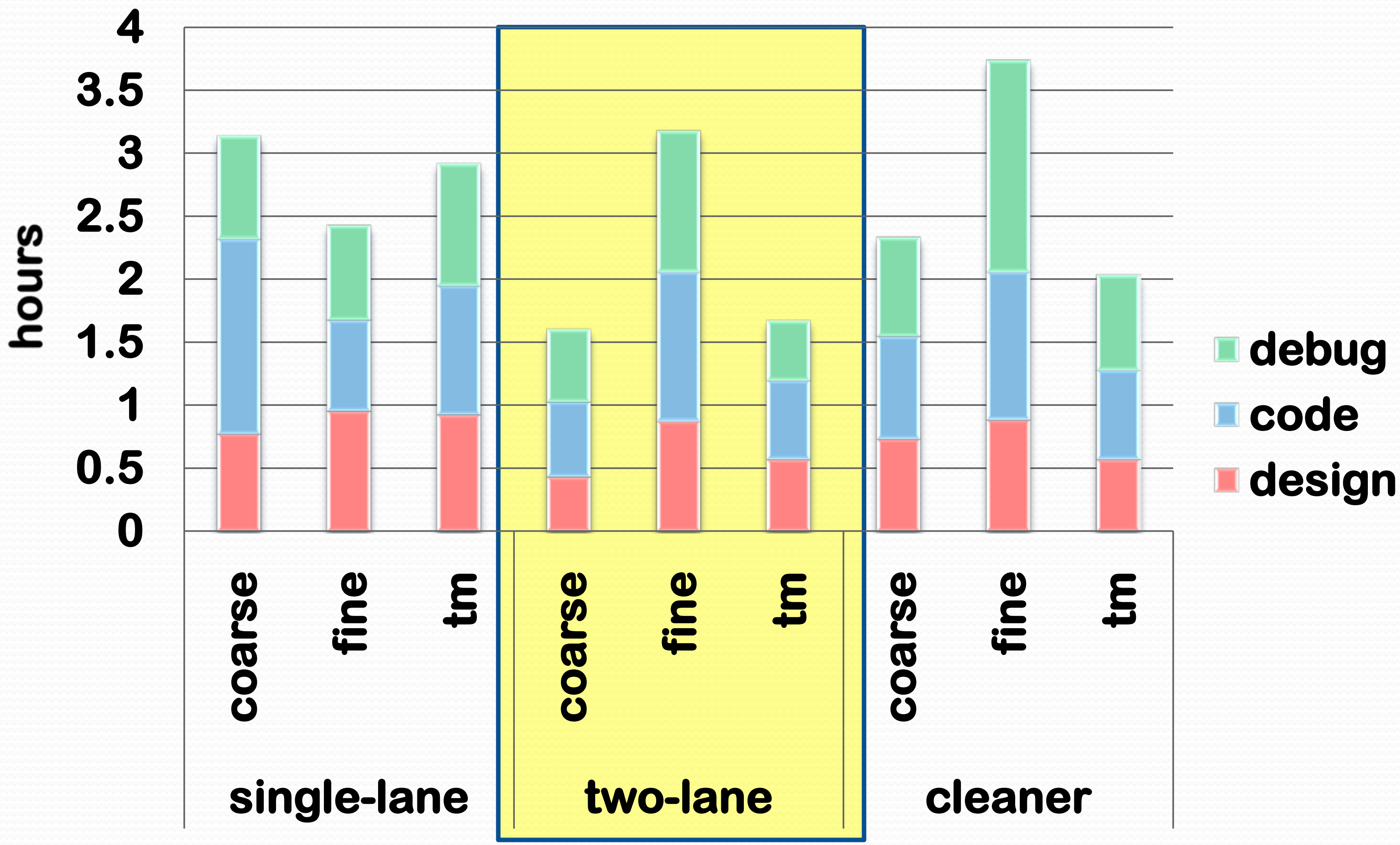
Development Effort: year 2



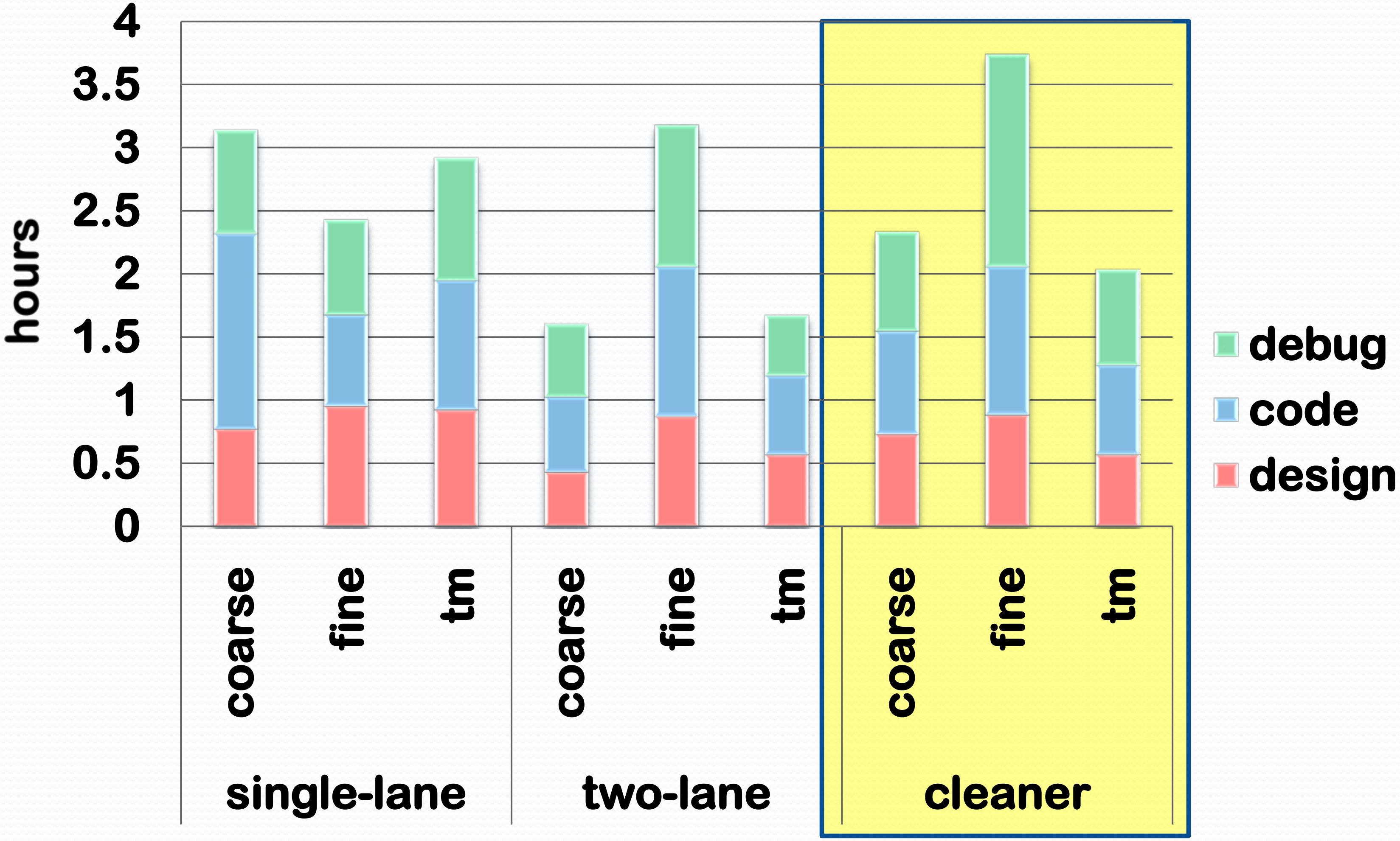
Development Effort: year 2



Development Effort: year 2



Development Effort: year 2



Qualitative preferences: year 2

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%**

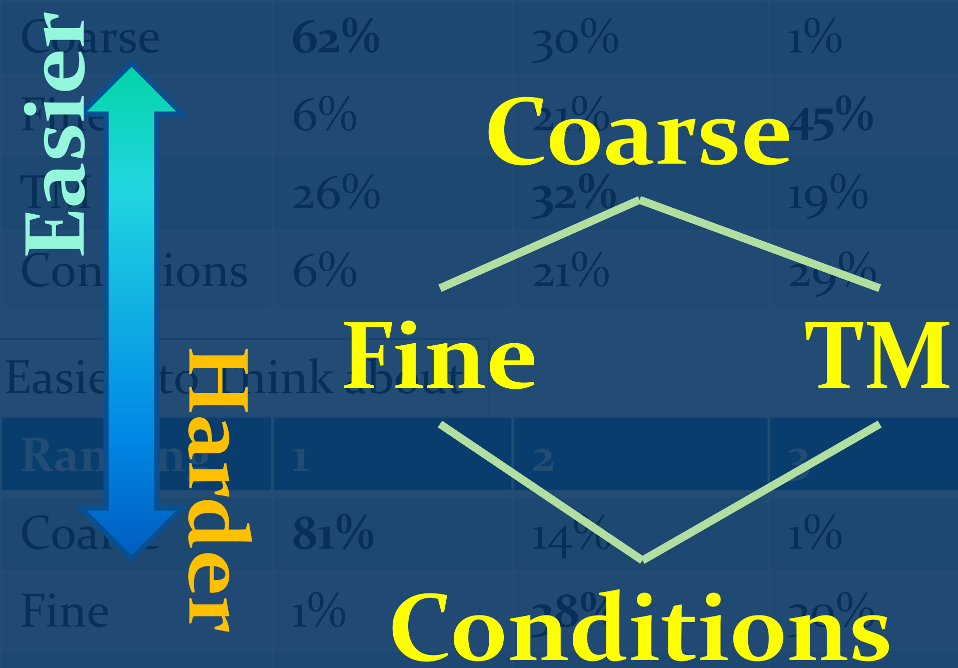
Easiest to Think about

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)

Qualitative preferences: year 2

Easiest to Think about



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%	28%	30%	29%
TM	16%	32%	30%	21%
Conditions	4%	14%	**40%**	**40%**

(Year 2)

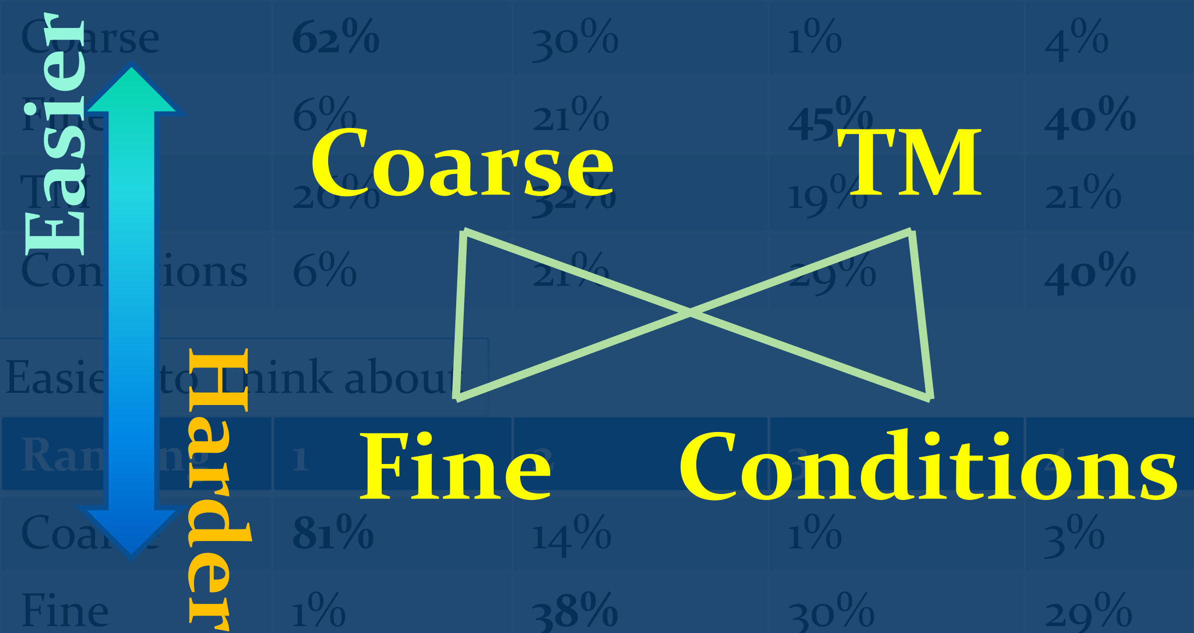
Qualitative preferences: year 2

Best Syntax

Ranking	1	2	3	4
Coarse	**62%**	30%	1%	4%
Fine	6%	21%	**45%**	**40%**
TM	20%	32%	19%	21%
Conditions	6%	21%	29%	**40%**

Easier 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%**



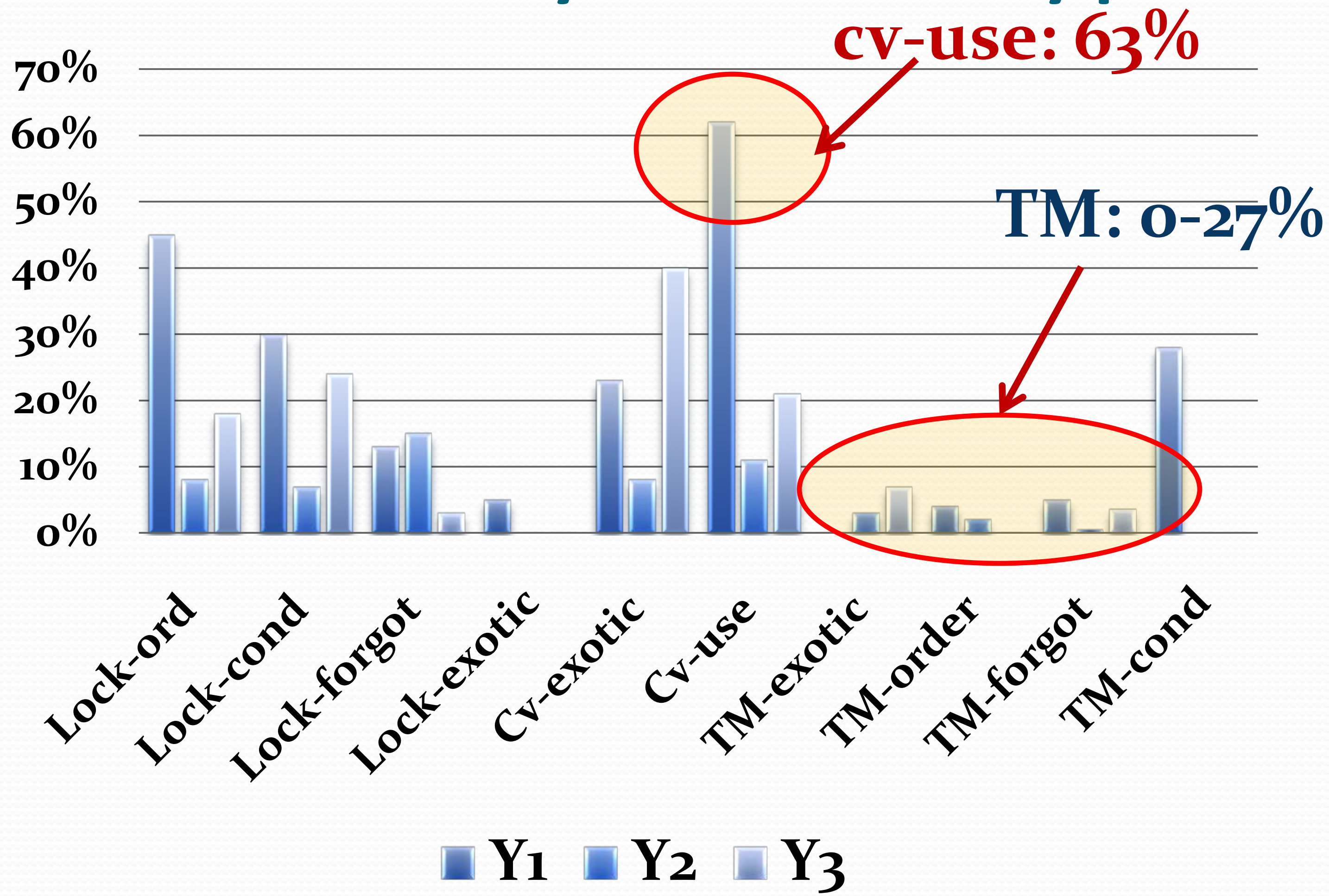
(Year 2)

Analyzing Programming Errors

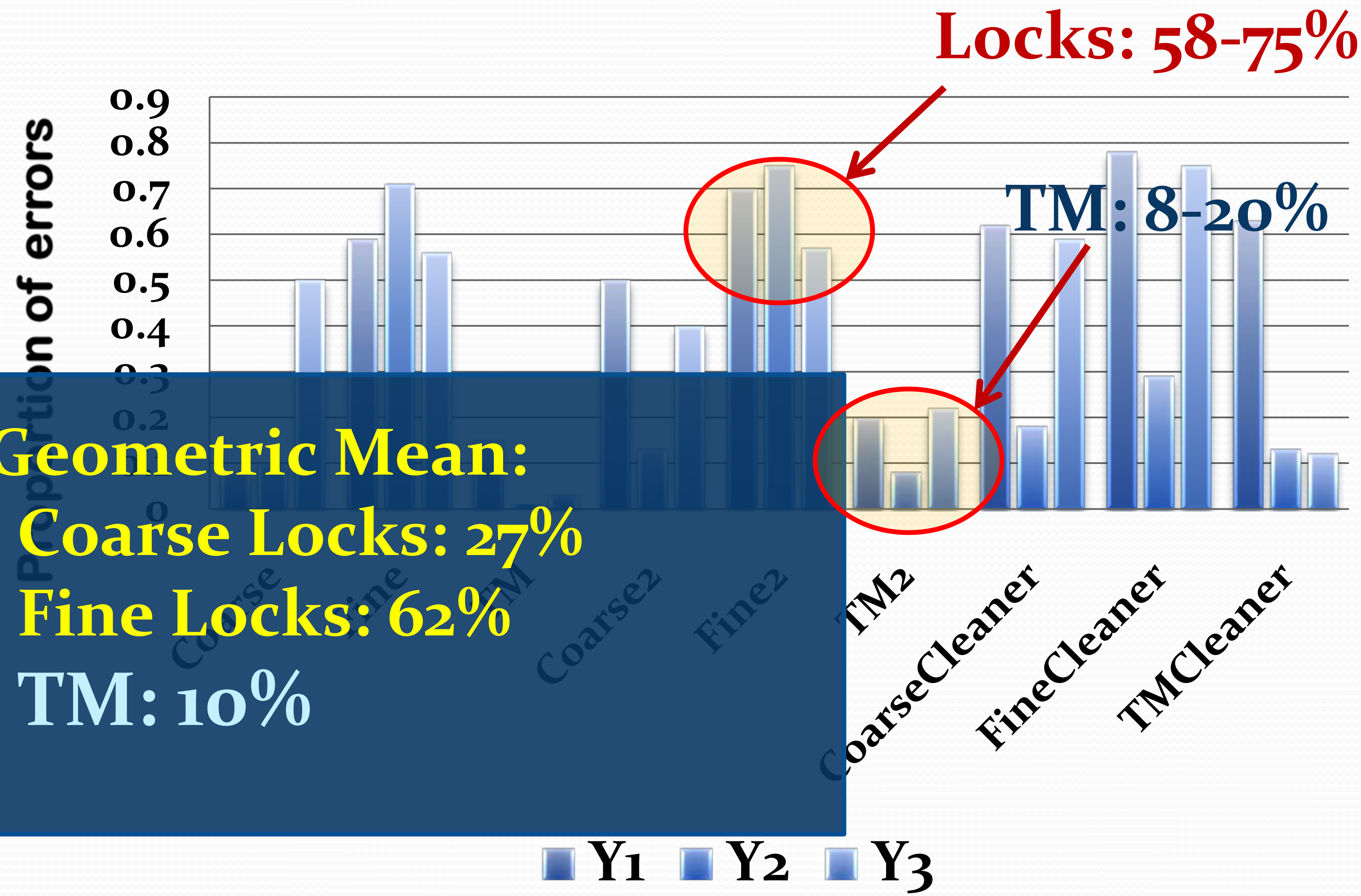
Error taxonomy: 10 classes

- **Lock-ord:** lock ordering
- **Lock-cond:** checking condition outside critsec
- **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 critsec
- ***TM-order:*** *Ordering in TM*

Error Rates by Defect Type



Overall Error Rates



Geometric Mean:

Coarse Locks: 27%

Fine Locks: 62%

TM: 10%

Outline

- Motivation
- Programming Problem
- User Study Methodology
- Results
- Conclusion

Conclusion

- General qualitative ranking:
  1. Coarse-grain locks (easiest)
  2. TM
  3. Fine-grain locks/conditions (hardest)
- Error rates overwhelmingly in favor of TM
- TM may ***actually be easier***

Overall Error Rates: Year 2