Autolocker: Synchronization Inference for Atomic Sections*

Bill McCloskey, Feng Zhou
UC Berkeley

David Gay
Intel Research

Eric Brewer
UC Berkeley

POPL 2006

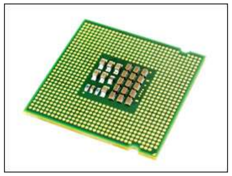
*In creating this presentation, I also used the authors' slides.

Outline

- Introduction
  - overview
  - benefits
- Autolocker algorithm
  - match locks to data
  - order lock acquisitions
  - insert lock acquisitions
- Related work
- Experimental evaluation
- Conclusions

Introduction

- Multi-core CPUs are here



- Concurrent programming is:
  - Difficult to reason about,
  - Prone to races and deadlocks.
- We need:
  - Simpler programming models,
  - Safer programs.

Autolocker: Overview

- Solution: pessimistic atomic sections
  - Why atomic?
    - Simplicity
    - Modularity
    - Safeness
  - Why pessimistic?
    - Compatibility
    - Less overhead than optimistic
- Implementation: intermediate tool that transforms atomic sequences to lock semantics

Autolocker: Overview

- Shared data is protected by annotated locks.
- Threads access shared data in atomic sections:

```
mutex m;
int shared_var protected_by(m);
atomic { ... x = shared_var; ... }
```

In an atomic section, code runs as if there is no concurrency.

- Threads never deadlock (due to Autolocker).
- Threads never race for protected data.

Autolocker Transformation

- Autolocker is a source-to-source transformation.

Autolocker code

```
mutex m1, m2;
int x protected_by(m1);
int y protected_by(m2);

atomic {
  y = 3;
  x = 2;
}
```

Suppose

⇨

m1 < m2.

C code

```
int m1, m2;
int x, y;

begin_atomic();
  acquire(m1);
  acquire(m2);
  y = 3;
  x = 2;
end_atomic();
```

Locks are acquired in a global order “<“ determined by partial orders. They are released when the outermost atomic section ends.

Granularity and Autolocker

- In Autolocker, annotations control performance:

```
struct entry {int x; char y;}
mutex m;
struct entry array[SIZE] protected_by(m);
```

⇩

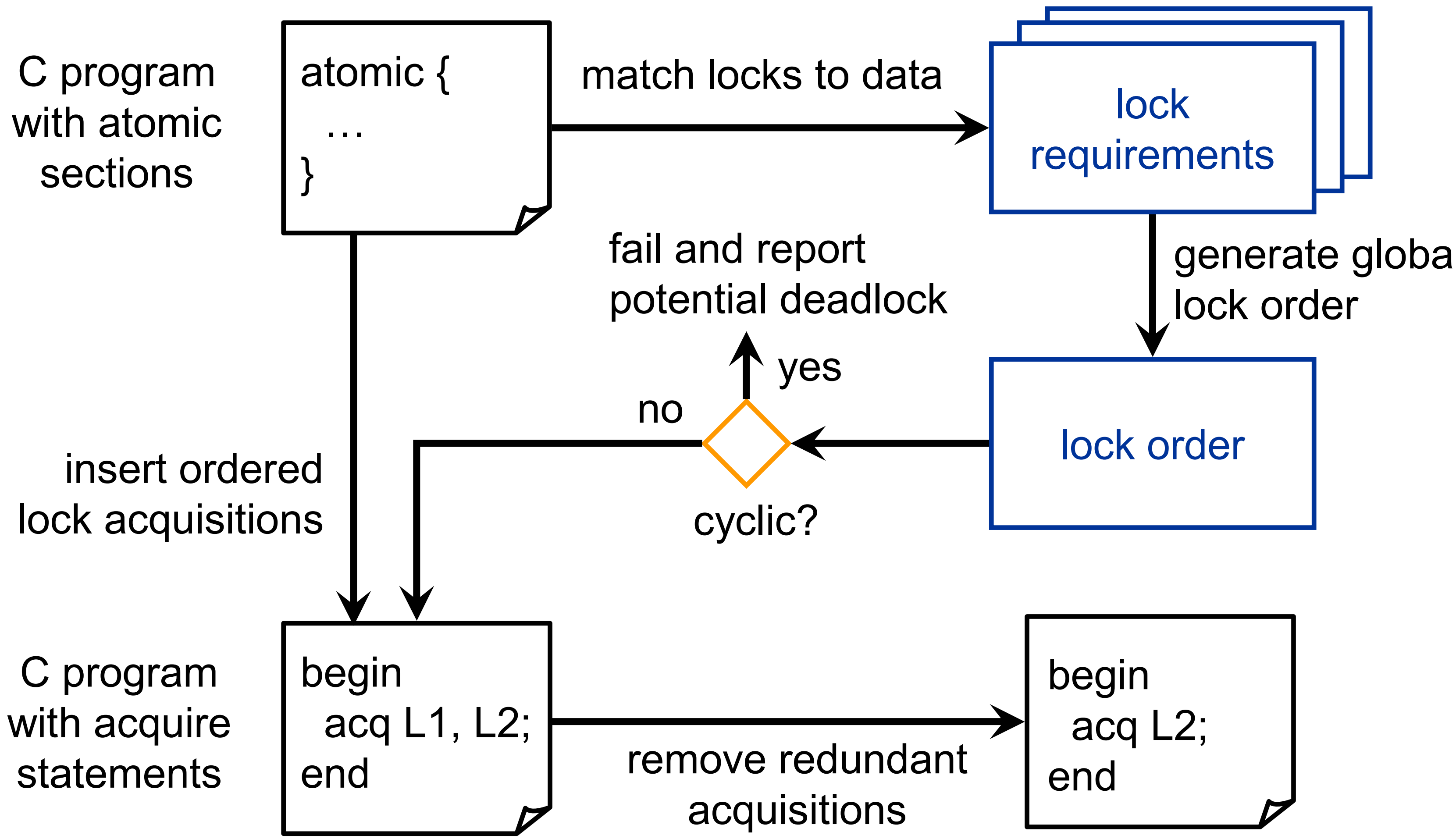
```
struct entry {mutex m; int x protected_by (m);
char y;}
struct entry array[SIZE];
```

- Simpler than redoing most of the locking
- *Changing annotations will not introduce deadlock or race conditions.*

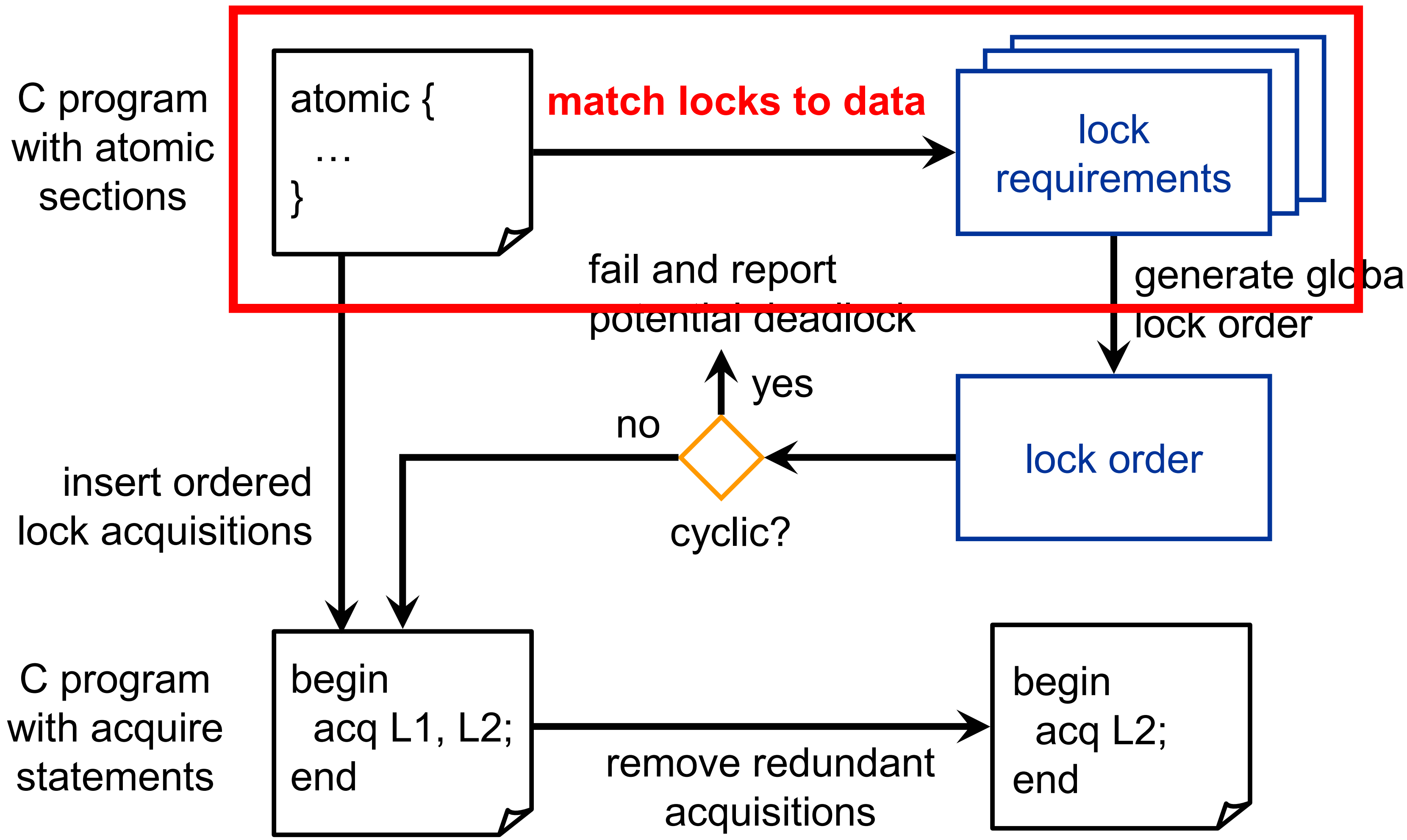
Outline

- Introduction
  - overview
  - benefits
- Autolocker algorithm
  - match locks to data
  - order lock acquisitions
  - insert lock acquisitions
- Related work
- Experimental evaluation
- Conclusion

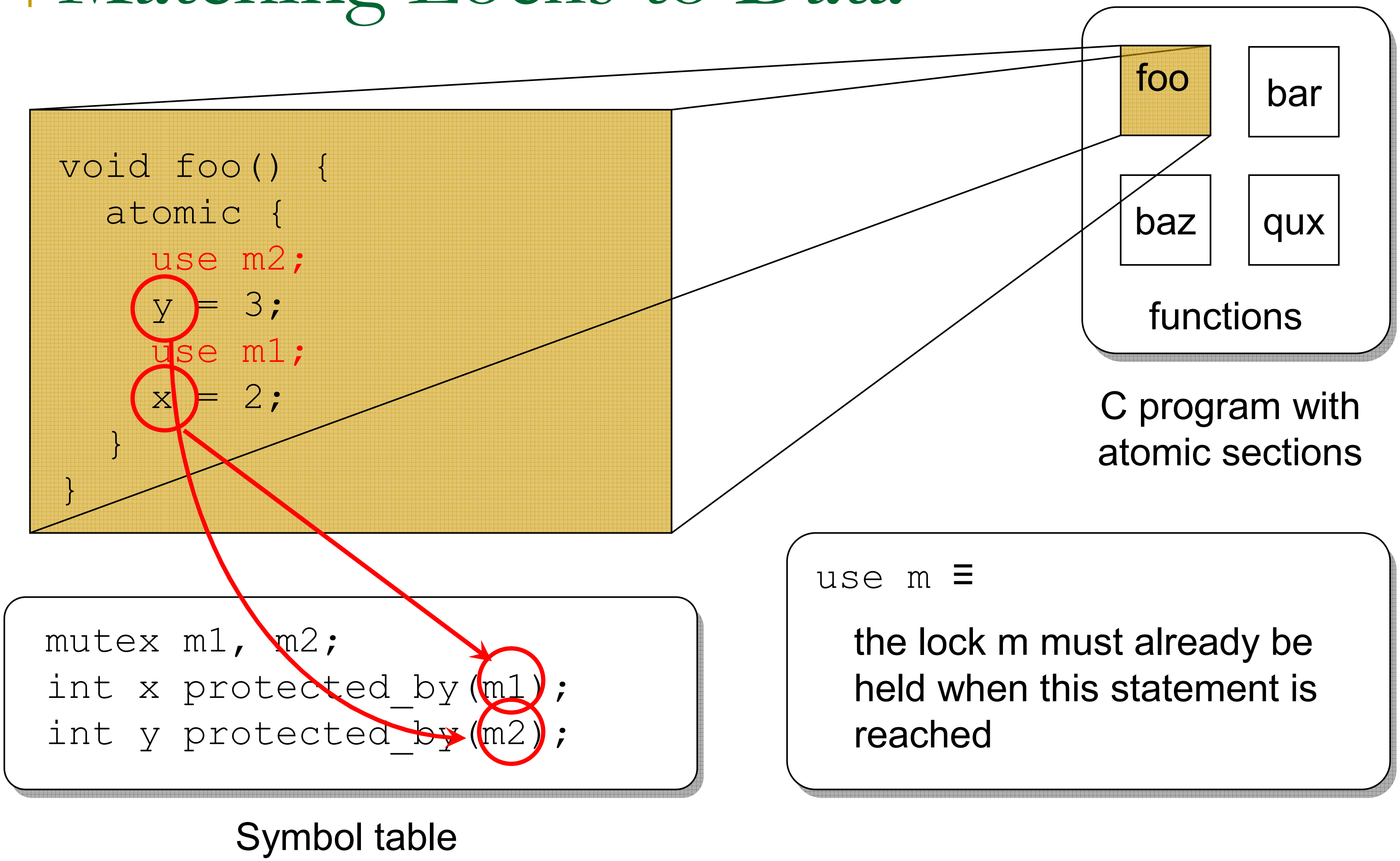
Algorithm Summary



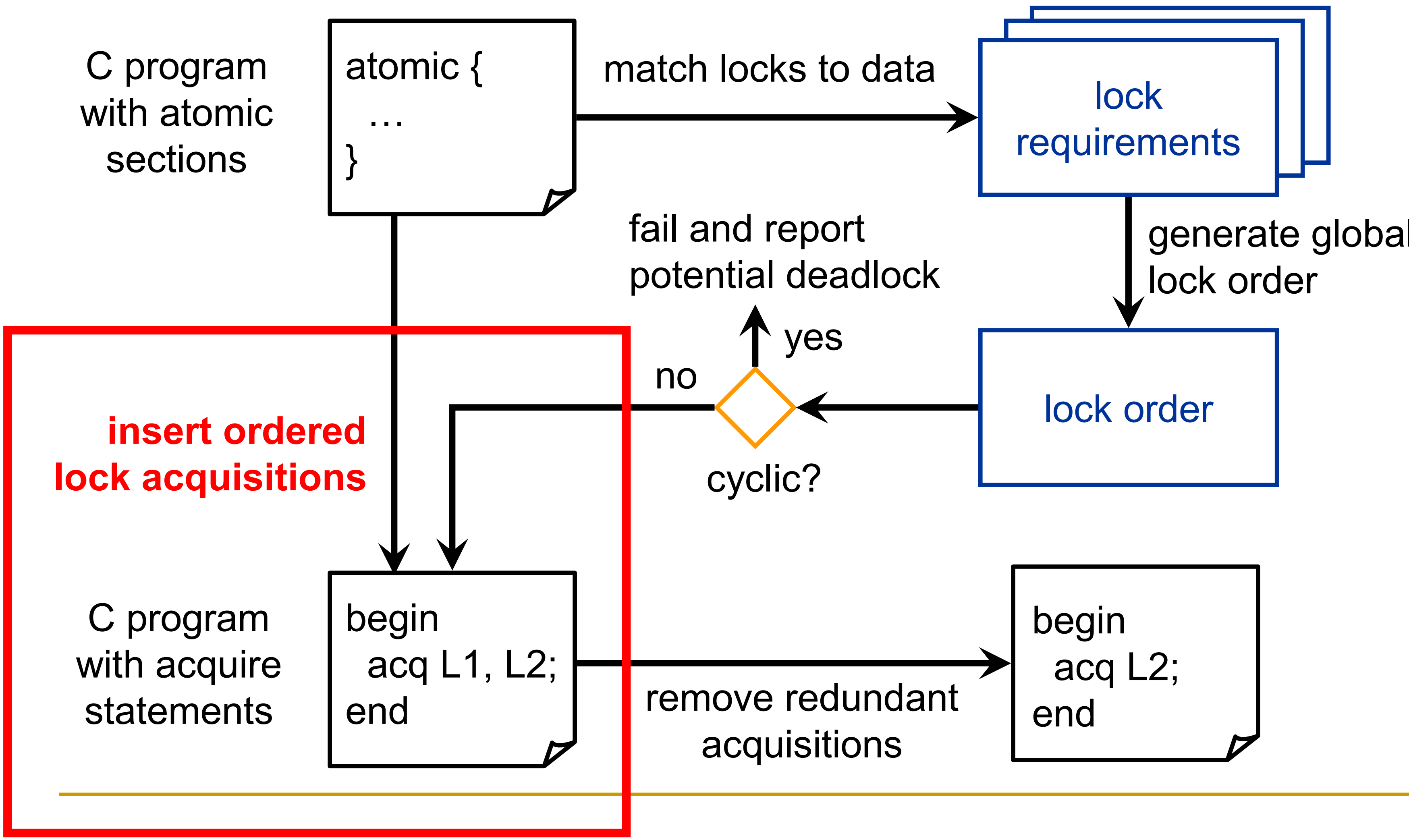
Algorithm Summary



Matching Locks to Data



Algorithm Summary



Acquisition Placement

- Assume there's an acyclic order "<" on locks

```
void foo() {
  atomic {
    acquire m1;
    acquire m2;
    y = 3;
    acquire m1;
    x = 2;
  }
}
```

We must acquire m2…

... but since m1 is needed later

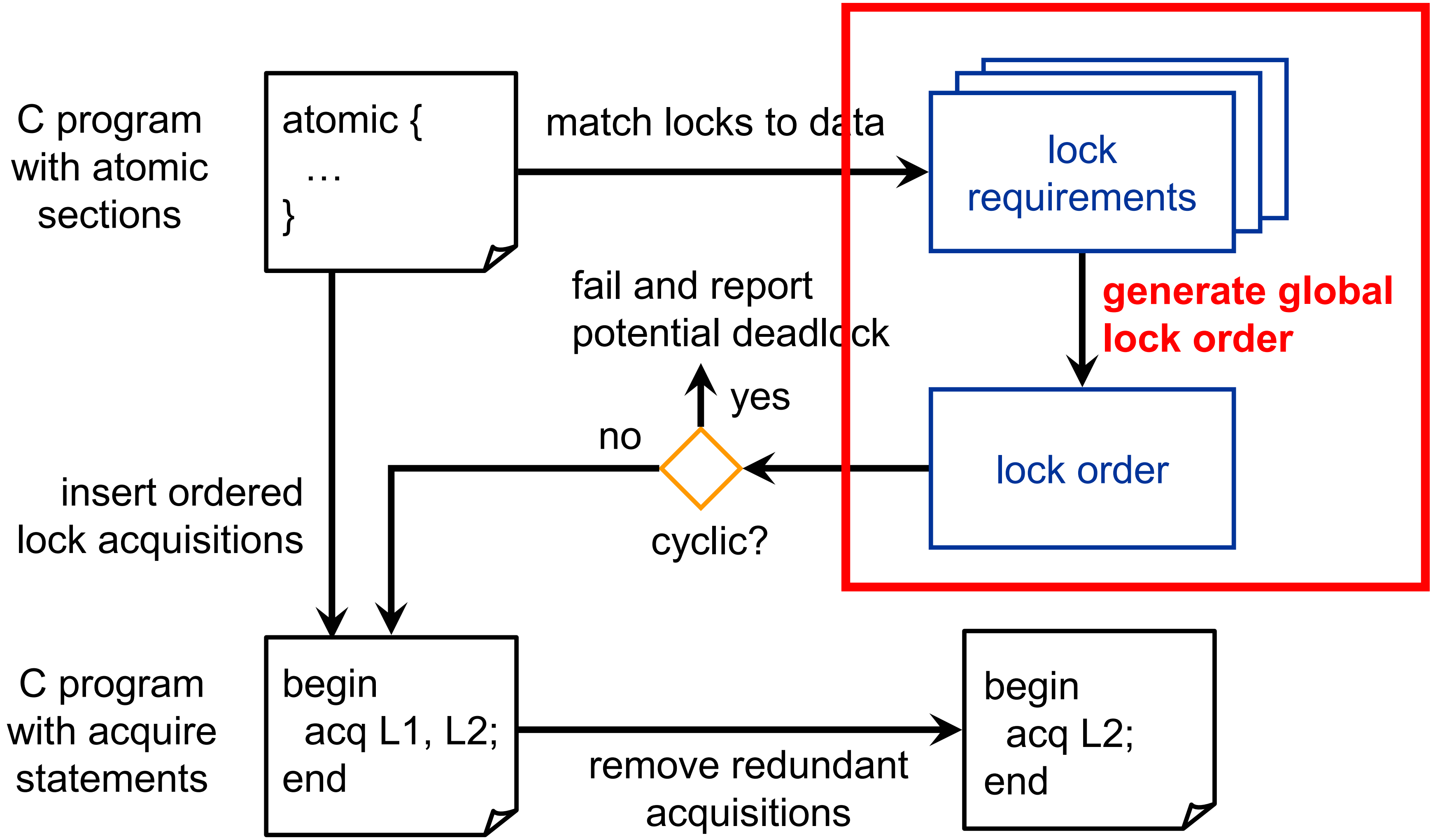
… and it is, in order, before m2

… we acquire m1 first

```
m1 < m2
```

global lock order

Algorithm Summary



Finding the Partial orders

- We search for any code matching this pattern:

```
use lock m1;
…
maykill lock m2;
…
use lock m2;
```

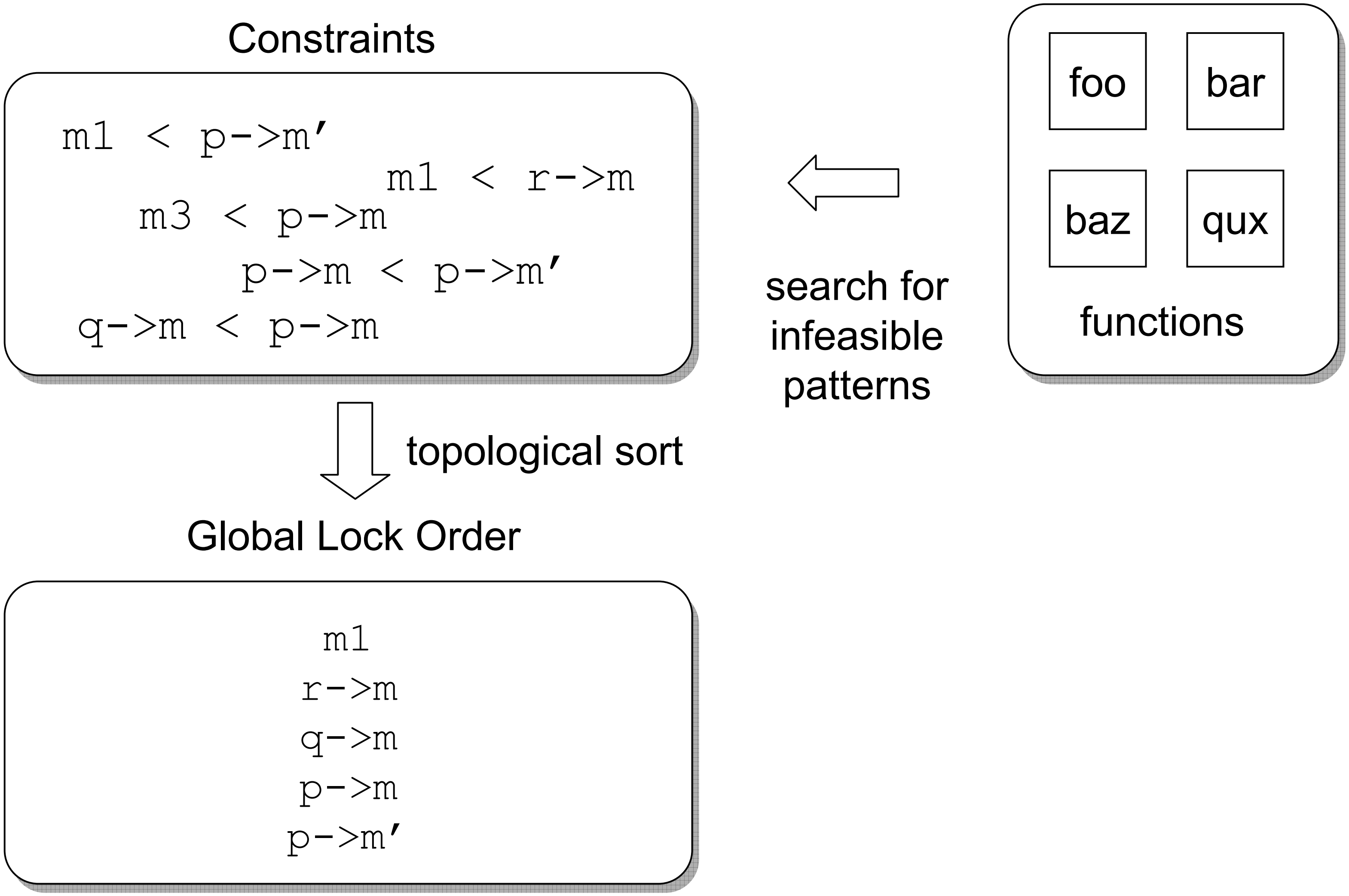
• “maykill lock m” happens when a lock is being assigned another value
• a lock cannot be acquired again after it was killed.
• a variable protected by a lock m cannot be accessed after m is killed if m is not acquired after the kill.

- Any feasible order must ensure `m1 < m2`

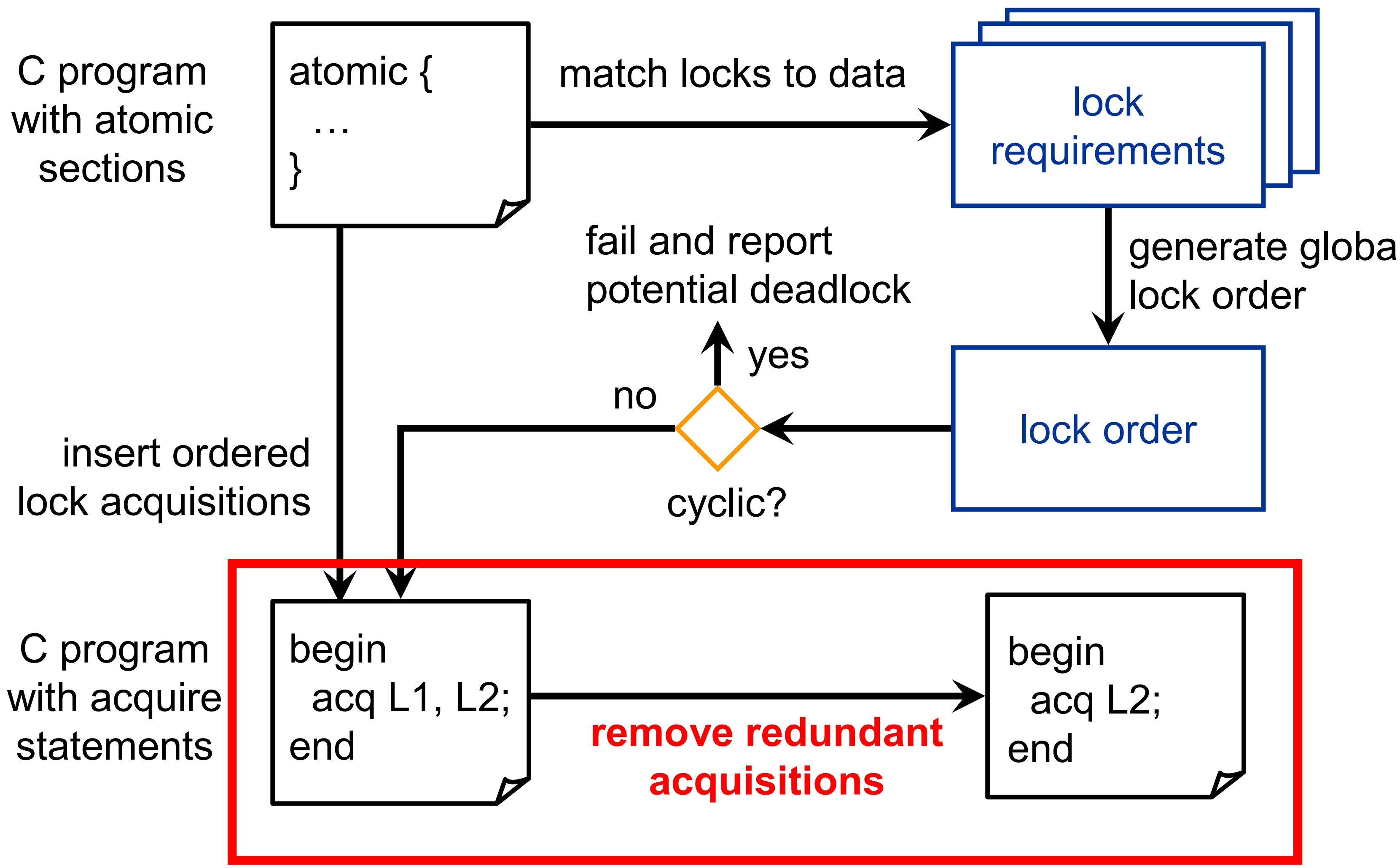
A burning question…

- The reason for which they allow locks to be overwritten is to give control over granularity to the programmer.
- This has drawbacks:
  - Create the entire partial order problem.
  - Such a global order might not exist.
  - Limits greatly expressiveness of language.
- Is the control over granularity really worth all these? Or, can we find a better solution?

Computing the Global Order



Algorithm Summary



Outline

- Introduction
  - overview
  - benefits
- Autolocker algorithm
  - computing protections
  - lock ordering
  - acquisition placement
- Related work
- Experimental evaluation
- Conclusion

Related Work

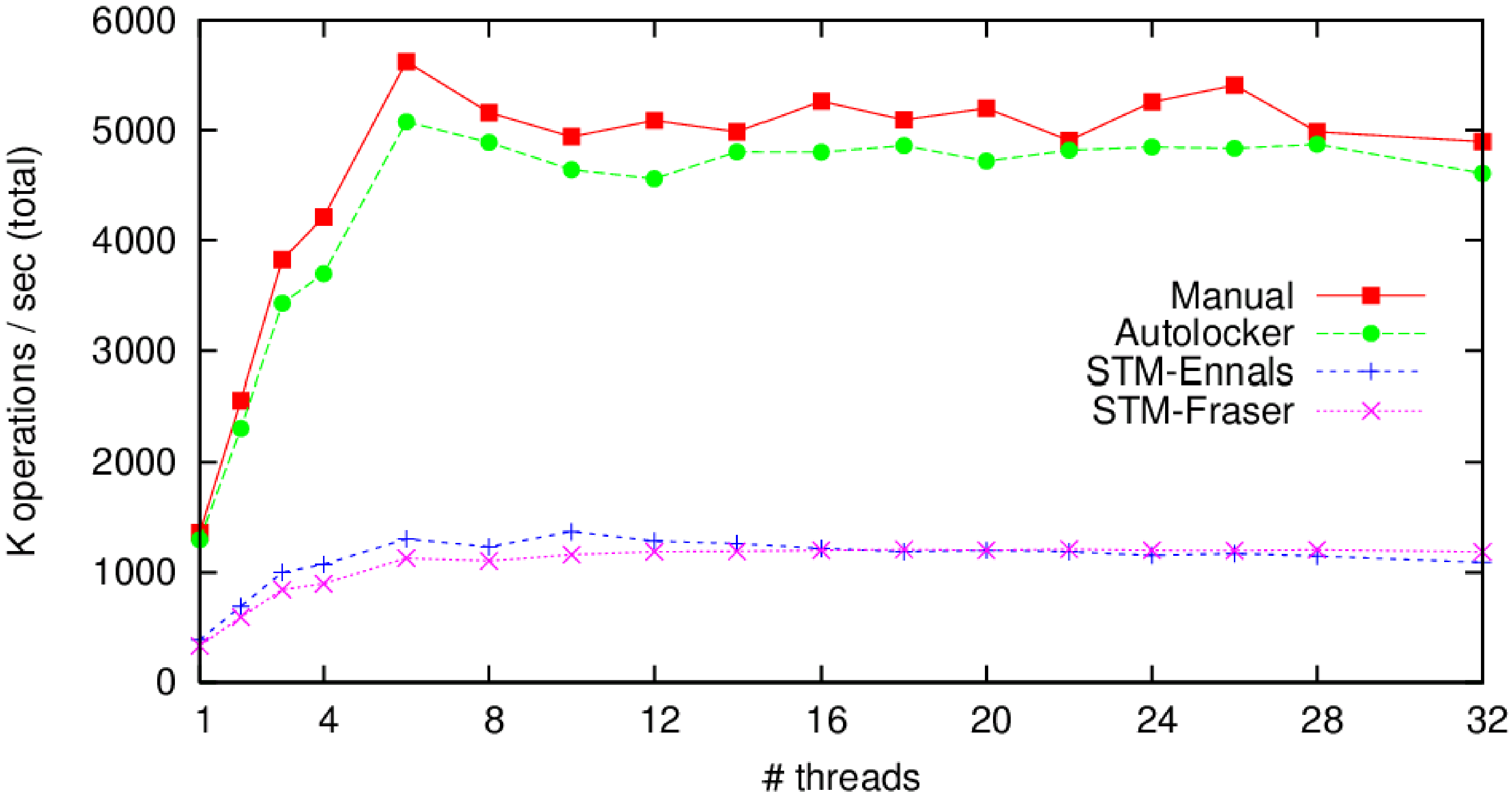
- *Transactional memory* also does atomic blocks
- Threads work locally and commit when done
- A thread rolls back if another thread changed the data it accessed
- Benefit: no complex static analysis
- Drawbacks:
  - software versions: can be slow
  - hardware versions: need new hardware
  - both: some operations cannot be rolled back (e.g., I/O)
- How does this compare with Autolocker?

Concurrent Hash Table

- Simple microbenchmark

- Compared Autolocker to:
  - manual locking
  - Fraser’s object-based transactional memory manager
  - Ennals’ revised transactional memory manager

Hash Table Benchmark



Machine: four processors, 1.9 GHz, HyperThreading, 1 GB RAM
Each datapoint is the average of 4 runs after 1 warmup run

AOL Server

- Threaded web server using manual locking

Size	52,589 lines
	82 modules
Changes	143 atomic sections added
	126 types annotated with protections
Problems	78/82 modules used original locking policies
Performance	negligible impact (~3%)

Outline

- Introduction
  - overview
  - benefits
- Autolocker algorithm
  - computing protections
  - lock ordering
  - acquisition placement
- Related work
- Experimental evaluation
- Conclusion

Conclusions

- Contributions:
  - a new model for programming parallel systems
  - a transformation tool to implement it
- Benefits:
  - performance close to well-written manual locking
  - freedom from deadlocks
  - freedom from races on protected data

My Conclusions

- Besides annotations, programmers have to be aware of rules
- Decreased expressiveness
- Two-phase locking has limitations
- I think it is a good start if we can avoid overwriting locks…

- Questions?