Lightweight Analyses For Reliable Concurrency

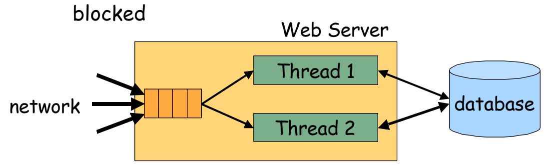
Stephen Freund
Williams College (currently on leave at UCSC)

joint work with Cormac Flanagan (UCSC) and
Shaz Qadeer (MSR)

1

Programming With Threads

- Decompose program into pieces that can run in parallel
- Advantages
  - exploit multiple processors
  - threads make progress, even if others are blocked

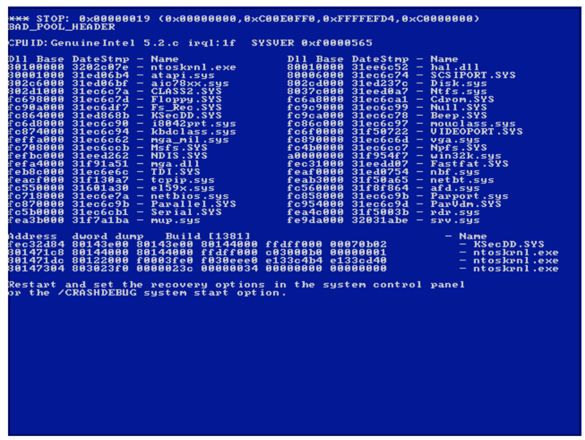


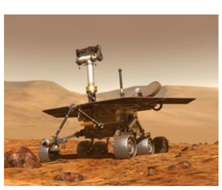
2

Towards Reliable Multithreaded Software

- Multithreaded software
  - increasingly common (Java, C#, GUIs, servers)
  - trends will continue
    - multi-core chips
- Heisenbugs due to thread interference
  - race conditions
  - atomicity violations

3





5

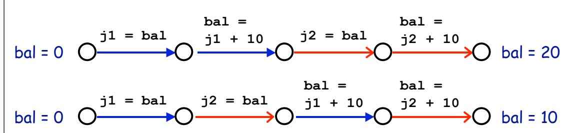
Bank Account Implementation

```
class Account {
  private int bal = 0;

  public void deposit(int n) {
    int j = bal;
    bal = j + n;
  }
}
```

7

Bank Account Implementation

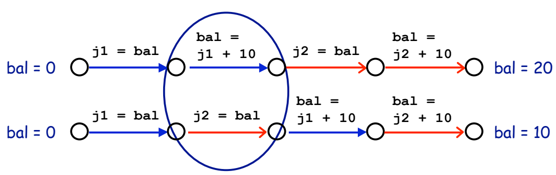
```
class Account {
  private int bal = 0;

  public void deposit(int n) {
    int j = bal;
    bal = j + n;
  }
}
```



8

Bank Account Implementation

A *race condition* occurs if two threads access a shared variable at the same time, and at least one of the accesses is a write



9

Race-Free Bank Account

```
class Account {
  private int bal = 0;

  public void deposit(int n) {
    synchronized(this) {
      int j = bal;
      bal = j + n;
    }
  }
}
```


10

Race-Free Bank Account

```
class Account {
  private int bal = 0;

  public int read() {
    int j;
    synchronized(this) {
      j = bal;
    }
    return j;
  }

  public void deposit(int n) {
    int j = read();
    // other thread can update bal
    synchronized(this) {
      bal = j + n;
    }
  }
}
```

11

Optimized Bank Account

```
class Account {
  private int bal = 0;

  public int read() {
    return bal;
  }

  public void deposit(int n) {
    synchronized(this) {
      int j = bal;
      bal = j + n;
    }
  }
}
```

12

Race-Freedom

- Race-freedom is neither necessary nor sufficient to ensure the absence of errors due to unexpected interactions between threads
- Is there a more fundamental semantic correctness property?

13

Atomicity

- A method is atomic if concurrent threads do not interfere with its behavior
- Informally, a method behaves the same regardless of what else is happening

14

Motivations for Atomicity

1. Stronger property than absence of data races
   - bad race-free programs
   - good "racy" programs

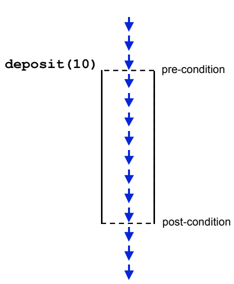
15

Motivations for Atomicity

1. Stronger property than absence of data races
2. Enables sequential reasoning

16

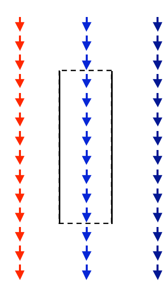
Sequential Program Execution




```
void deposit(int n) {
  ...
}
```

17

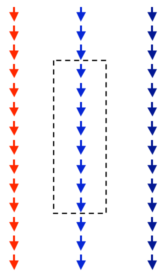
Multithreaded Execution



```
void deposit(int n) {
  ...
}
```

18

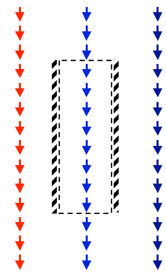
Multithreaded Execution



```
void deposit(int n) {
  ...
}
```

19

Multithreaded Execution



- Guarantees concurrent threads do not interfere with atomic method
- Enables sequential reasoning

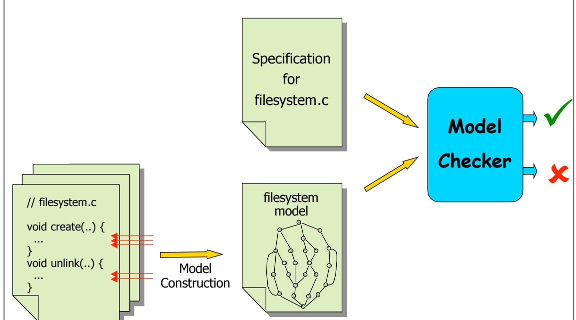
20

Motivations for Atomicity

1. Stronger property than absence of data races
2. Enables sequential reasoning
3. Simple specification

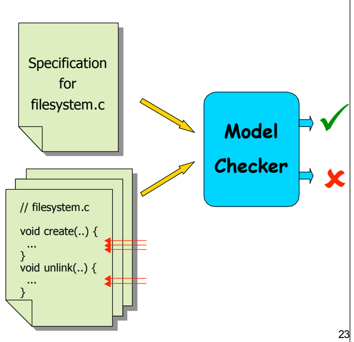
21

Model Checking of Software Models



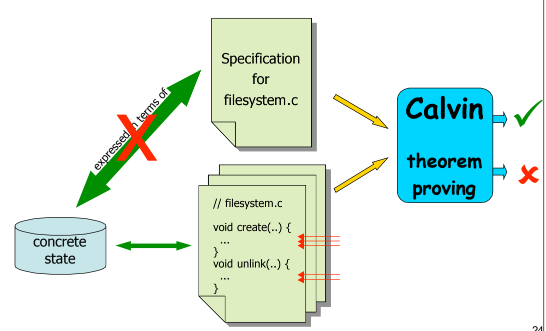
22

Model Checking of Software



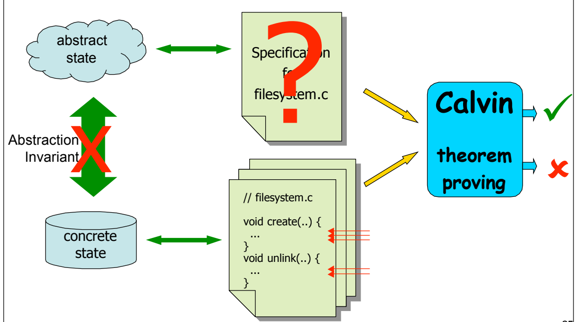
23

Experience with Calvin Software Checker



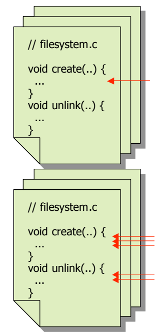
24

Experience with Calvin Software Checker



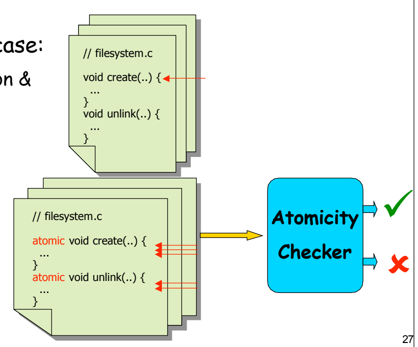
The Need for Atomicity

Sequential case:
code inspection &
testing mostly ok



The Need for Atomicity

Sequential case:
code inspection &
testing ok

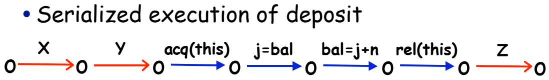


Motivations for Atomicity

1. Stronger property than absence of data races
2. Enables sequential reasoning
3. Simple specification

Atomicity

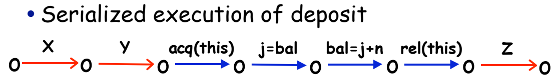
- Serialized execution of deposit



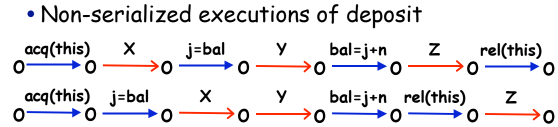

```
class Account {
  int bal;
  void deposit(int n) {
    synchronized (this) {
      int j = bal;
      bal = j + n;
    }
  }
}
```


Atomicity

- Serialized execution of deposit



- Non-serialized executions of deposit



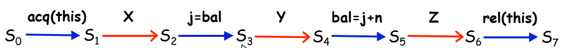
- deposit is *atomic* if, for every non-serialized execution, there is a serialized execution with the same overall behavior

Atomicity

- Canonical property
  - (linearizability, serializability, ...)
- Enables sequential reasoning
  - simplifies validation of multithreaded code
- Matches practice in existing code
  - most methods (>80%) are atomic
  - many interfaces described as "thread-safe"
- Can verify atomicity statically or dynamically
  - violations often indicate errors
  - leverages Lipton's theory of reduction

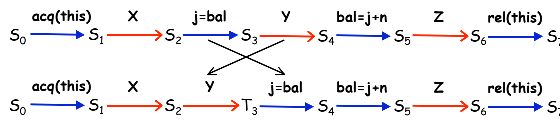
Reduction [Lipton 75]

Reduction [Lipton 75]

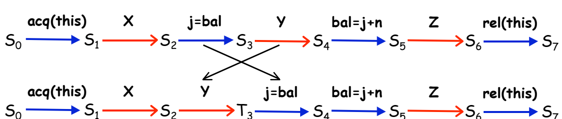


blue thread holds lock
⇒ red thread does not hold lock
⇒ operation y does not access balance
⇒ operations commute

Reduction [Lipton 75]

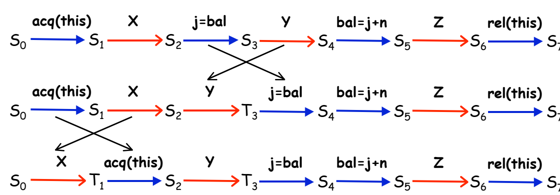


Reduction [Lipton 75]

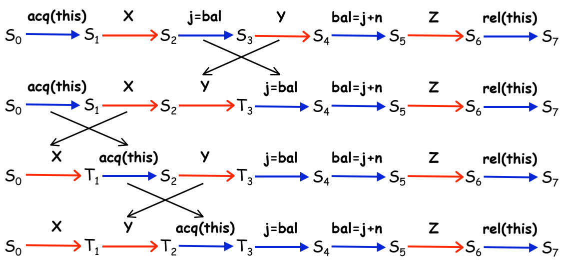


blue thread holds lock after acquire
⇒ operation x does not modify lock
⇒ operations commute

Reduction [Lipton 75]

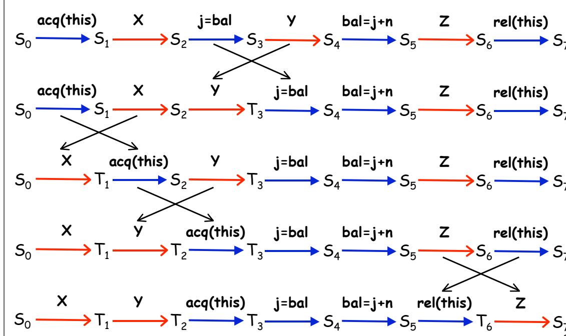


Reduction [Lipton 75]



37

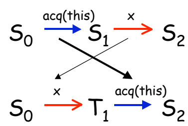
Reduction [Lipton 75]



38

Movers

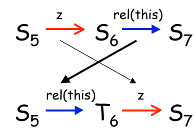
- right-mover
  - lock acquire



39

Movers

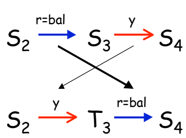
- right-mover
  - lock acquire
- left-mover
  - lock release

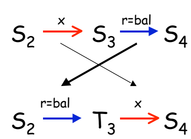


40

Movers

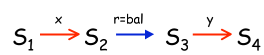
- right-mover
  - lock acquire
- left-mover
  - lock acquire
- both-mover
  - race-free field access





41

Movers



- right-mover
  - lock acquire
- left-mover
  - lock acquire
- both-mover
  - race-free field access
- non-mover (atomic)
  - access to "racy" fields

42

Code Classification

right:	lock acquire
left:	lock release
both-mover:	race-free variable access
atomic:	conflicting variable access

acq(this) → right, j=bal → both, bal=j+n → both, rel(this) → left

atomic

- reducible blocks have form: (right|both)*[atomic](left|both)*

43

Composing Atomicities

```
void deposit(int n) {
   int j;
   synchronized(this) { j = bal; }
   synchronized(this) { bal = j + n; }
}
```

acq(this) → right, j=bal → both, rel(this) → left, acq(this) → right, bal=j+n → both, rel(this) → left

atomic   atomic

compound

44

java.lang.StringBuffer

```
/**
  ... used by the compiler to implement the binary
  string concatenation operator ...

  String buffers are safe for use by multiple
  threads. The methods are synchronized so that
  all the operations on any particular instance
  behave as if they occur in some serial order
  that is consistent with the order of the method
  calls made by each of the individual threads
  involved.
*/
/*# atomic */ public class StringBuffer {
  ...
}
```

FALSE

45

java.lang.StringBuffer

```
/*# atomic */ public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }

  public synchronized void append(StringBuffer sb){

    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(...,len,...);
    ...
  }
}
```

sb.length() acquires lock on sb, gets length, and releases lock

other threads can change sb

use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)

46

java.lang.StringBuffer

```
/*# atomic */ public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }

  public synchronized void append(StringBuffer sb){

    int len = sb.length();            A
    ...
    ...
    ...
    sb.getChars(...,len,...);         A
    ...
  }
}
```

Compound

47

Tools for Checking Atomicity

- Calvin-R: ESC for multithreaded code (2 KLOC)
  - [Freund-Qadeer 03]
- A type system for atomicity (20 KLOC)
  - [Flanagan-Qadeer 03,Flanagan-Freund-Lifshin 05]
- Atomizer dynamic atomicity checker (200 KLOC)
  - [Flanagan-Freund 04]

http://www.cs.williams.edu/~freund/atom.html

48

Calvin-R

```
/* global_invariant (\forall int i; inodeLocks[i] == null ==> 0 <= inodeBlocknos[i] && inodeBlocknos[i] < Daisy.MAXBLOCK) */

//@ requires 0 <= inodenum && inodenum < Daisy.MAXINODE;
//@ requires i != null
//@ requires DaisyLock.inodeLocks[inodenum] == \tid
//@ modifies i.blockno, i.size, i.used, i.inodenum
//@ ensures i.blockno  == inodeBlocknos[inodenum]
//@ ensures i.size     == inodeSizes[inodenum]
//@ ensures i.used     == inodeUsed[inodenum]
//@ ensures i.inodenum == inodenum
//@ ensures 0 <= i.blockno && i.blockno < Daisy.MAXBLOCK
static void readi(long inodenum, Inode i) {
        i.blockno = Petal.readLong(STARTINODEAREA +
                                        (inodenum * Daisy.INODESIZE));
        i.size= Petal.readLong(STARTINODEAREA +
                                  (inodenum * Daisy.INODESIZE) + 8);
        i.used = Petal.read(STARTINODEAREA +
                                 (inodenum * Daisy.INODESIZE) + 16) == 1;
        i.inodenum = inodenum;
        // read the right bytes, put in inode
} //@ nowarn Post
```

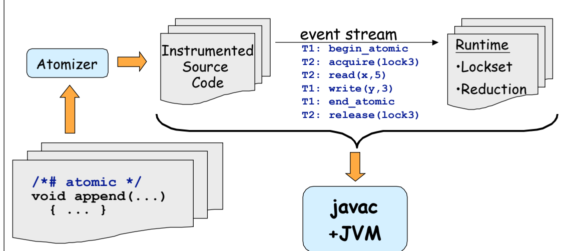

Lightweight Tools For Atomicity

- Part 1
  - Runtime analysis
  - practical aspects of building / validating tools
- Part 2
  - Type systems for concurrency and atomicity
- Part 3
  - Beyond reduction
  - "Purity", abstraction, commit-atomicity, ...

Atomizer: Documenting Atomicity

- Manual annotations
  - `/*# atomic */ void append(...) { ... }`
- Heuristics
  - all synchronized blocks are atomic
  - all public methods are atomic, except main and run
  - these heuristics are very effective

Atomizer: Instrumentation Architecture



Warning: method "append"
may not be atomic at line 43

55

Lockset Algorithm [Savage et al 97]

- Tracks *lockset* for each field
  - lockset = set of locks held on all accesses to field
- Dynamically infers protecting lock for each field
  - empty lockset indicates possible race condition
- Reduction algorithm leverages race information

56

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;      ⇨
  }
  o.f = 11;
}
```

Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

- First access to **o.f**:

LockSet(**o.f**) := Held(curThread)

= { x, y }

57

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;       ⇨
}
```

Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

- Subsequent access to **o.f**:

LockSet(**o.f**) := LockSet(**o.f**) ∩ Held(curThread)

= { x, y } ∩ { x } = { x }

58

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

Thread 2

```
synchronized(y) {
  o.f = 2;        ⇦
}
```

- Subsequent access to **o.f**:

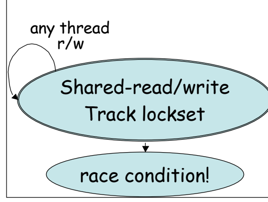
LockSet(**o.f**) := LockSet(**o.f**) ∩ Held(curThread)

= { x } ∩ { y } = { }

RACE CONDITION!

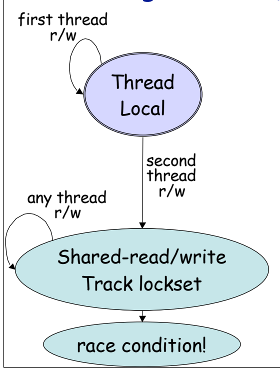
59

Lockset



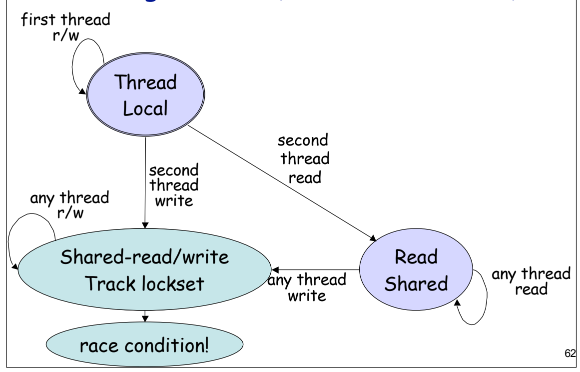
60

Extending Lockset (Thread Local Data)



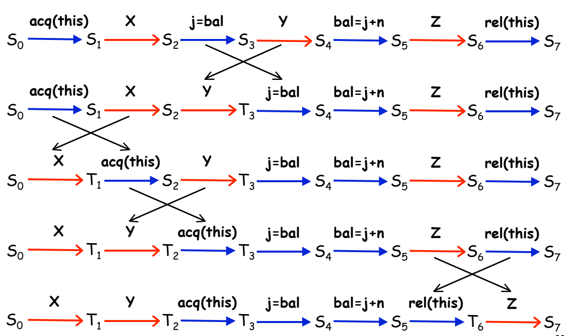
61

Extending Lockset (Read Shared Data)



62

Reduction [Lipton 75]

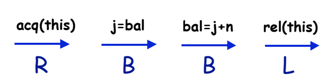


63

Movers

- R: right-mover
  - lock acquire
- L: left-mover
  - lock release
- B: both-mover
  - race-free field access
- A: atomic
  - access to "racy" fields

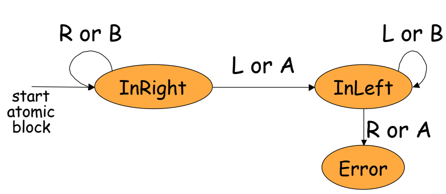
•Reducible blocks have form (R|B)* [A] (L|B)*



64

Dynamic Reduction

- Automata to check (R|B)* [A] (L|B)*



- Track state per atomic block in dynamic scope with stack of automata

65

Reporting Errors

```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
  /*# atomic */
  public synchronized void append(StringBuffer sb){

    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(...,len,...);
    ...
  }
}
```

sb.length() acquires lock on sb, gets length, and releases lock

other threads can change sb

use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)

66

Reporting Errors

```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
  /*# atomic */
  public synchronized void append(StringBuffer sb){

    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(...,len,...);
    ...
  }
}
```

```
StringBuffer.append is not atomic:
  Start:
    at StringBuffer.append(StringBuffer.j
    at Thread1.run(Example.java:17)

  Commit: Lock Release
    at StringBuffer.length(StringBuffer.j
    at StringBuffer.append(StringBuffer.j
    at Thread1.run(Example.java:17)

  Error: Lock Acquire
    at StringBuffer.getChars(StringBuffer
    at StringBuffer.append(StringBuffer.j
    at Thread1.run(Example.java:17)
```

67

Atomizer Review

- Instrumented code calls Atomizer run time
  - on field accesses, sync ops, etc
- Lockset algorithm identifies races
  - used to classify ops as movers or non-movers
- Atomizer checks reducibility of atomic blocks
  - warns about atomicity violations

68

Refining Race Information

- Discovery of races during reduction

```
      /*# atomic */
      void deposit(int n) {
R       synchronized (this) {
B         int j = bal;
          // other thread changes bal
A         bal = j + n;
L       }
```

69

Extensions

- Redundant lock operations
  - acquire is right-mover
  - release is left-mover
  - Want to treat them as both movers when possible
- Write-protected data
  - common idiom

70

Thread-Local Data

```
class Vector {
  atomic synchronized Object get(int i) { ... }
  atomic synchronized void add(Object o) { ... }
}

class WorkerThread {

  atomic void transaction() {
    Vector v = new Vector();
    v.add(x1);
    v.add(x2);
    ...
    v.get(i);
  }
}
```

71

Reentrant Locks

```
class Vector {

  atomic synchronized Object get(int i) { ... }
  atomic synchronized Object add(Object o) { ... }

  atomic boolean contains(Object o) {
    synchronized(this) {
      for (int i = 0; i < size(); i++)
        if (get(i).equals(o)) return true;
    }
    return false;
  }
}
```

72

Layered Abstractions

```
class Set {
  Vector elems;

  atomic void add(Object o) {
    synchronized(this) {
      if (!elems.contains(o)) elems.add(o);
    }
  }
}
```


Redundant Lock Operations

- Acquire is right-mover
- Release is left-mover

- Redundant lock operations are both-movers
  - acquiring/releasing a thread-local lock
  - re-entrant acquire/release
  - acquiring/releasing lock A, if lock B always acquired before A

Write-Protected Data

```
class Account {
    int bal;
    /*# atomic */ int read() { return bal; }
    /*# atomic */ void deposit(int n) {
R     synchronized (this) {
B       int j = bal;
A       bal = j + n;
L     }
    }
}
```

- Lock `this` held whenever balance is updated
  - write must hold lock, and is non-mover
  - read without lock held is non-mover
  - read with lock held is both-mover

Extending Lockset for Write-Prot Data

- Track *access* lockset and *write* lockset
  - access lockset = locks held on every access
  - write lockset = locks held on every write
- For regularly-protected data
  - access lockset = write lockset = { protecting lock }
- For write-protected data
  - access lockset = Ø
  - write lockset = { write-protecting lock }
- Read is both-mover if at least one *write* lock held
- Write is both-mover if *access* lockset not empty

Evaluation

- 12 benchmarks
  - scientific computing, web server, std libraries, ...
  - 200,000+ lines of code

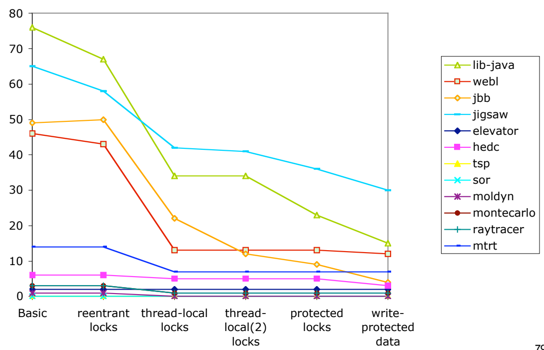
- Heuristics for atomicity
  - all synchronized blocks are atomic
  - all public methods are atomic, except `main` and `run`

- Slowdown: 1.5x - 45x

Performance

Benchmark	Lines	Base Time (s)	Slowdown
elevator	500	11.2	-
hedc	29,900	6.4	-
tsp	700	1.9	21.8
sor	17,700	1.3	1.5
moldyn	1,300	90.6	1.5
montecarlo	3,600	6.4	2.7
raytracer	1,900	4.8	41.8
mtrt	11,300	2.8	38.8
jigsaw	90,100	3.0	4.7
specJBB	30,500	26.2	12.1
webl	22,300	60.3	-
lib-java	75,305	96.5	-

Extensions Reduce Number of Warnings



Evaluation

- Warnings: 97 (down from 341- extensions necessary!)
- Real errors (conservative): 7
- False alarms due to:
  - simplistic heuristics for atomicity
    - programmer should specify atomicity
  - false races
  - methods irreducible yet still "atomic"
    - eg caching, lazy initialization (more later)
- No warnings reported in more than 90% of exercised methods

Example Bugs

```
class PrintWriter {
    Writer out;
    public void println(String s) {
      synchronized(lock) {
        out.print(s);
        out.println();
      }
    }
 }

class ResourceStoreManager {
    synchronized checkClosed() { ... }
    synchronized lookup(...) { ... }
    public ResourceStore loadResourceStore(...) {
      checkClosed();
      return lookup(...);
    }
 }
```


Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - types [Flanagan-Qadeer 03], model checking [Stoller-Cohen 03, Flanagan-Qadeer 03], procedure summaries [Qadeer et al 04]
- Other atomicity checkers
  - [Wang-Stoller 03], Bogor model checker [Hatcliff et al 03]
  - view consistency [Artho-Biere-Havelund 03, von Praun-Gross 03]
- Race detection / prevention
  - dynamic [Savage et al 97, O'Callahan-Choi 01, von Praun-Gross 01]
  - Warlock [Sterling 93], SPMD [Aiken-Gay 98]
  - type systems [Abadi-Flanagan 99, Flanagan-Freund 00, Boyapati-Rinard 01, Grossman 03]
  - Guava [Bacon et al 01]

Software Transactions

- Language support for lightweight transactions
  - [Fraser-Harris 03], [Harris-Marlow-Jones-Herlihy 05]
- Example:

```
transaction (x > 10) {
  x = x - 10;
}
```

- Run-time waits until condition is true and then executes body "atomically"
  - no programmer-inserted concurrency control

Naive Lock-Based Implementation

- Acquire global lock when entering transaction, release lock when exiting
- Can try to use more fine-grained locking
  - hard to scale
  - hard to do automatically
- Database solutions?

Optimistic Concurrency

- Run transaction as normal code
- Log reads/writes to shared variables (but do not modify them)
- On commit, check whether interference has occurred
  - have shared variables been modified?
  - if so, discard log and retry
  - if not, commit logged changes
- STM (Software Transactional Memory)
  - run-time support for tracking shared variables, modification history, etc.

85

Transactions vs. Atomicity

- Orthogonal techniques
  - programmer vs. run-time control
- Transactions avoid:
  - deadlocks, priority inversion, ...
- Can transactions scale?
  - overhead, retry rate, non-"undoable" ops
  - large transactions
- Combining atomicity and transactions
  - optimize transactions
  - focus on most critical performance bottlenecks

86

Atomizer Summary

- Atomicity
  - enables sequential analysis
  - matches practice
- Improvements over race detectors
  - catches "higher-level" concurrency errors
  - some benign races do not break atomicity
- Next steps
  - expressiveness
  - hybrid tools

87

Looking Ahead

- Limitations of Atomizer
  - coverage
  - whole program
  - annotating large code bases
- Static type systems
  - modular checking
  - inferring specifications
  - computational / expressiveness issues

88