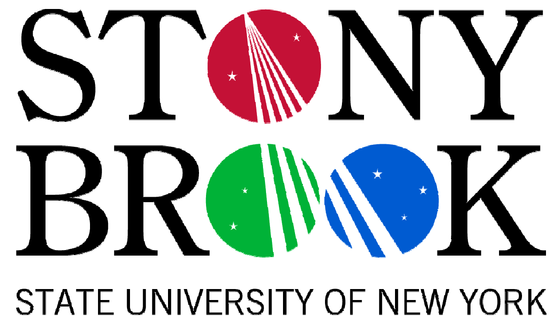
Checking Atomicity in Concurrent Java Programs

Scott D. Stoller

Joint work with Rahul Agarwal, Amit Sasturkar, & Liqiang Wang



Example: An Atomicity Error

```
public class Vector extends ... implements ... {
    int elementCount;
    Object[] elementData;
    public Vector(Collection c) {
        elementCount = c.size();
        elementData = new Object[(int)Math.min((elementCount*110L)/100,
                                                Integer.MAX_VALUE)];
        c.toArray(elementData);
    }
    public synchronized int size() { return elementCount; }
    public synchronized Object[] toArray(Object a[]) { ... }
    public synchronized void removeAllElements() { ... }
    public synchronized boolean add(Object o) { ... }
}
```

Another thread may execute v1.removeAllElements() or v1.add(o) here.

One thread executes: Vector v2= new Vector(v1);

Outcome: v2 may be full of null elements (behavior of toArray with wrong size argument). No exception is thrown.

Definition of Atomicity

- A code block is **atomic** if every execution of the program is equivalent to an execution in which the code block is executed without interruption (by other threads).
- **Example:** The green code is atomic if an execution like

is equivalent to one of the following executions

- Atomicity is an important correctness requirement.
- Atomicity makes subsequent analysis faster.

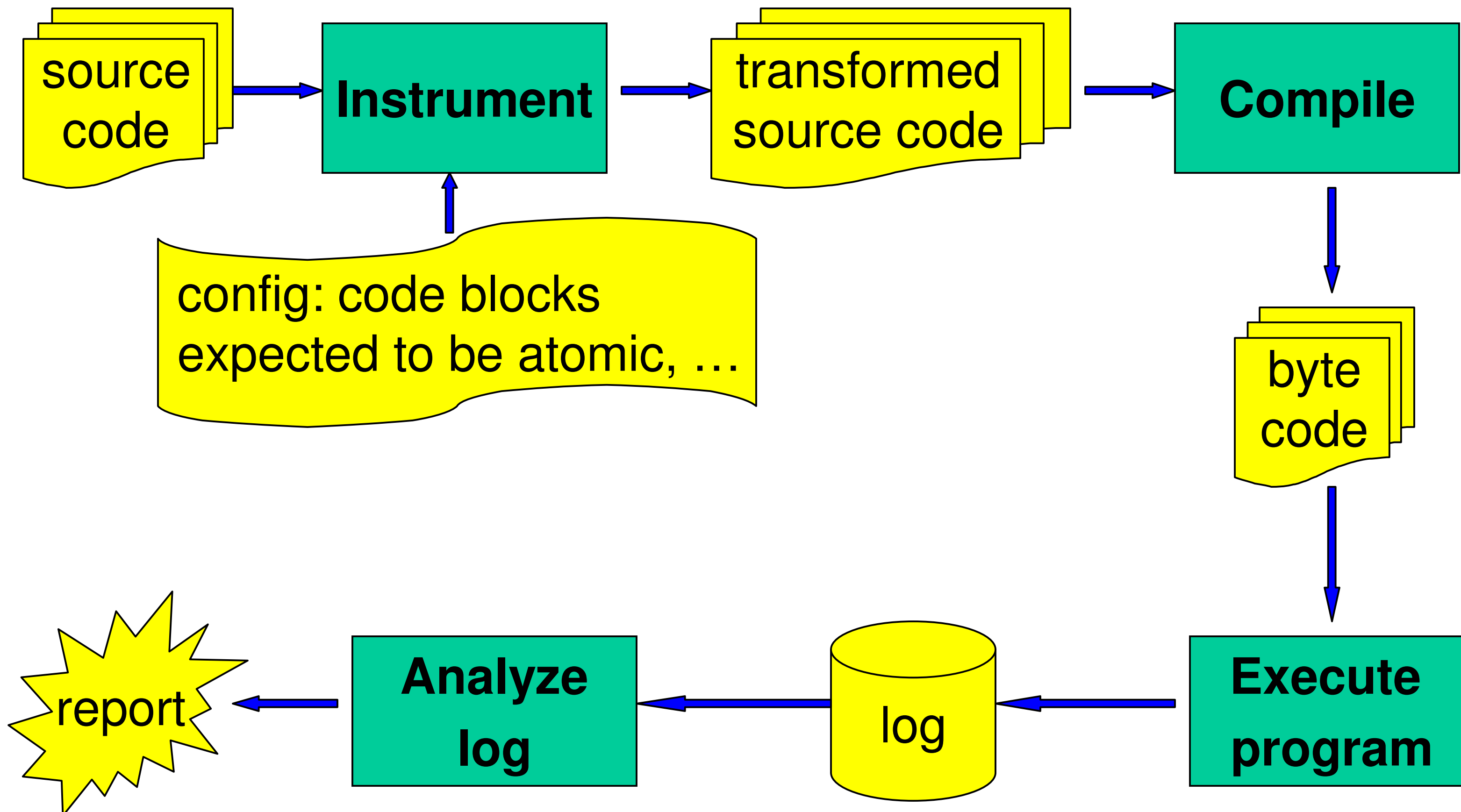
Atomicity and Serializability

- Atomicity is similar to serializability (isolation) of transactions.
  - In databases, the DBMS enforces a standard concurrency control (synchronization) policy on all transactions.
    - Example: 2-phase locking
  - A concurrent program may use multiple synchronization policies, and synchronization primitives are scattered throughout the program.

Outline

- Run-time Atomicity Checking
  - More scalable.
  - Does not guarantee atomicity in other runs.
- Atomicity Types (and Race-Free Types)
  - Guarantee atomicity in all runs of the program
  - Focus on type inference, for automatic analysis
- Optimized Run-time Atomicity Checking using Types

Run-time Atomicity Checking



Simple Run-time Atomicity Checking

- **Transaction:** sequence of events executed by a thread in a code block expected to be atomic
- An execution is **serial** if each transaction is executed without interruption by other threads.
- An execution is **serializable** if it is equivalent to some serial execution.

1. Record the execution.
2. Check whether it is serializable.
3. If not, report an atomicity violation.

Problem: Many observed executions are serial!

Effective Run-time Atomicity Checking

- Don't consider only the observed execution. Analyze the synchronization in it. Does the synchronization prevent non-serializable interleavings of transactions? If not, report a potential atomicity violation.
- Compared to the simple analysis:
  - Greatly increased effectiveness at finding defects
    - Can find potential atomicity violations in serial executions
  - False alarms are possible
    - Run-time analysis does not look at the program, so it can only guess which interleavings of events from different transactions are possible.

View Atomicity

- Stricter condition, easier to check, usually equivalent
- A set of transactions is **view atomic** if every feasible interleaving of them is view equivalent to some serial execution of them.
- Two executions are **view equivalent** if
  - Each read event has the same predecessor write event
  - The final write event to each shared variable is the same
- An interleaving is **feasible** if it is consistent with the synchronization in the transactions.
  - this is an approximation of actual feasibility
- "atomicity" = "view atomicity" in the rest of the talk

Example of View Atomicity

- {t1, t2} is not atomic. t1= rd(x) wr(x), t2 = wr(x).

All feasible interleavings:

E1: rd(x) wr(x) wr(x) serial (potential violation is reported)

E2: wr(x) rd(x) wr(x) serial (potential violation is reported)

E3: rd(x) wr(x) wr(x) not serial, not equivalent to E1 or E2

- {t3, t4} is atomic. t3= acq(*l*) rd(x) wr(x) rel(*l*)
  t4 = acq(*l*) wr(x) rel(*l*).

All feasible interleavings:

E4: acq(*l*) rd(x) wr(x) rel(*l*) acq(*l*) wr(x) rel(*l*) serial

E5: acq(*l*) wr(x) rel(*l*) acq(*l*) rd(x) wr(x) rel(*l*) serial

Atomicity Checking: Two Approaches

- **Reduction-Based Approach**
  - Analyze commutativity of events.
  - Lipton's reduction theorem provides a simple condition on commutativities that implies atomicity.
  - Efficient but conservative (may produce false alarms).
- **Block-Based Approach**
  - Analyze interleavings of blocks (small fragments) of different transactions
  - More precise, but more expensive, than reduction-based.
  - Much cheaper than the brute-force approach of analyzing interleavings of entire transactions.

Lipton-like Reduction Theorem

Theorem: A set T of transactions is atomic if

1. T has no potential for deadlock (details omitted), and
2. Each transaction in T has the form $R^* N^? L^*$.

R (right-mover): events *r* that right-commute, *i.e.*,

r e is equivalent to *e r* for all events *e* of other threads

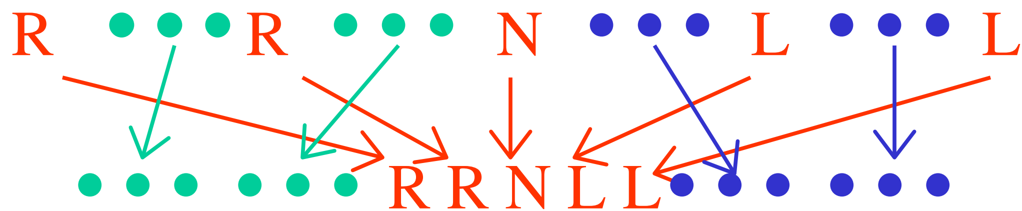
L (left-mover) : events that left-commute.

M (mover): R ∩ L

N (non-mover): events not known to commute.

Original Execution:

Equiv. Serial Execution:



Enhanced Reduction Theorem

Theorem: A set T of transactions is **atomic** if

1. T has no potential for deadlock, and
2. Each transaction in T has the form

   $(R + Acq\ B^*\ Rel)^*\ N^?\ (L + Acq\ B^*\ Rel)^*$.

Acq: lock acquire. $Acq \subseteq R$.

Rel : lock release. $Rel \subseteq L$.

B : accesses to read-only and thread-local variables.

$B \subseteq R$. $B \subseteq L$.

Which Accesses to Variables are Movers?

- **Recall:** A **race** occurs when two threads concurrently access a variable, and at least one of the accesses is a write.
  - Races typically indicate non-determinism.
  - Races are rare in most programs.
- An access is **race-free** if it cannot be involved in a race.
- **Theorem:** Race-free accesses are movers.
  - **Proof:** An adjacent operation by another thread cannot be a conflicting access to the same variable (because that would be a race), so the two operations commute.
- How to determine which accesses are race-free?

Run-time Race Detection

- **Lockset algorithm** [Savage+ 1997] detects potential races.
  - LkSet(v) = set of locks that protected variable v so far,
    i.e., locks held on all accesses to v so far
  - Initialization: LkSet(v) := set of all locks
  - On an access to v by a thread t,

    LkSet(v) := LkSet(v) ∩ locksHeld(t)
  - If LkSet(v) is empty, issue warning: potential race
- **Correctness:** If LkSet(v) is non-empty, there is no race on v; if there were, both threads involved must simultaneously hold the locks in LkSet(v), and that is impossible.

Extensions to the Lockset Algorithm

- **Dynamic Escape Analysis** (for object initialization)
  - Accesses to an object before it escapes (becomes reachable by other threads) cannot be involved in races.
  - [Savage+97] uses a simple but unsound heuristic.
- **Read-Only Variables** [Savage+97]
  - A variable that is only read after it escapes is race-free.
- **Start/Join Analysis**
  - If two threads do not run concurrently, operations in one cannot race with operations in the other.
- **Multi-Lockset Algorithm**
  - Multiple read locksets and a write lockset per variable

Reduction-Based Analysis of Atomicity

Step 1. Classify events. Lock acquire: R. Lock release: L. Race-free access (read or write): M. Other events: N.

Step 2. If a transaction does not match $R^* N^? L^*$, report a potential atomicity violation.

On-line Alg.: classify each access based on current lockset, etc.

Off-line Alg.: classify all accesses based on final lockset, etc.

Example: acq(*l*) rd(x) wr(x) rel(*l*) wr(x)

LkSet(x):	All	{*l*}	{*l*}	{*l*}	{ }	
On-line:	R	M	M	L	N	no atomicity warnings
Off-line:	R	N	N	L	N	red transact'n not atomic

Motivation for Block-Based Analysis

- Reduction-based analysis produces false alarms.
- **Example:** t1: rd(x) wr(x) t2: rd(x)
  - {t1, t2} is atomic. Reduction-based algorithm produces a false alarm, because all of the events are non-movers.
- **Block-Based Analysis:** more accurate (fewer false alarms)
- **Intuition:** Whenever two transactions are not atomic, there exist 3 or 4 accesses (plus synchronization) that are the root cause of the atomicity violation.
  - **Example:** t1: ••• rd(x)••• wr(x)••• t2: ••• rd(x) •••

    Looking at the 3 accesses shown explicitly is enough to see that {t1, t2} is not atomic.

Block-Based Analysis of Atomicity

- **Block for t:** two accesses (reads or writes) from t, with info about synchronization:
  - locks held at each access, locks held continuously from one access to the other, ...
- **Atomicity** of a set of blocks: analogous to atomicity of transactions (each block is like a small transaction).
- **Idea:** {t1, t2} is atomic iff for all blocks b1 for t1 and all blocks b2 for t2, {b1, b2} is atomic.
  - Check whether {b1, b2} is atomic by considering all (at most 6) feasible interleavings of them.

Example of Block-Based Analysis

- t1: acq(l) acq(m) • • • rd(x) rel(m) • • • rd(x) • • • rel(l)
- t2: • • • acq(m) • • • wr(x) • • • rel(m) • • • rd(x) • • •

block	trans	e1	e2	held(e1)	held(e2)	heldAcross
b1	t1	rd(x)	rd(x)	{l,m}	{l}	{l}
b2	t2	wr(x)	rd(x)	{m}	{ }	{ }

- {b1, b2} is not atomic.
  - The interleaving rd(x) wr(x) rd(x) is feasible and non-serializable.
- Therefore {t1, t2} is not atomic.

Refined Definition of Blocks

1v-block for t: two events e1 and e2 from t that access the same variable and such that

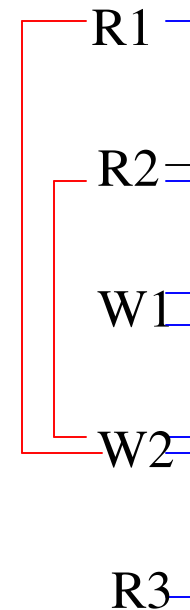
- if (there is write before e2)

  e1 is the last write which precedes e2;

  else

  e1 is the last read which precedes e2;
- if (e2 is the final write)

  e1 is an initial read.



Initial read in t: a read not preceded in t by a write to the same variable.

The number of 1v-blocks is O(#events).

Block-Based Analysis: Multiple Variables

- Analysis using 1v-blocks is insufficient for transactions that share multiple variables.
  - **Example:** rd(x) wr(x) rd(y) wr(y). Not atomic, but the degenerate 1v-blocks are trivially atomic.
- **2v-block for t:** two events in InitRd(t) $\cup$ FnlWr(t) that access different variables.
  - InitRd(t): **initial reads** in t, i.e., reads not preceded in t by a write to the same variable
  - FnlWr(t): **final writes** in t, i.e., writes not followed in t by another write to the same variable
- It suffices to use other accesses only in 1v-blocks.

Block-Based Analysis: Pairs of Transactions

- **Theorem:** {t1, t2} is atomic iff
  - For all 1v-blocks b1 for t1 and all 1v-blocks b2 for t2, {b1, b2} is atomic.
  - For all 2v-blocks b1 for t1 and all 2v-blocks b2 for t2, {b1, b2} is atomic.
- Our tool primarily uses this theorem to check that the transactions in an execution are pairwise atomic.
  - In theory, this does not imply that the set of all transactions in the execution is atomic (example soon).
  - In practice, it usually does (always in our experiments).

Block-Based Analysis: Multiple Transactions

- If all pairs of transactions in a set T are atomic, T might not be atomic, due to cyclic dependencies.
- **Example:** The set {wr(x) wr(y), rd(x) wr(z), rd(z) rd(y)} is not atomic, because of the non-serializable interleaving wr(x) rd(x) wr(z) rd(z) rd(y) wr(y), but all pairs of transactions in it are atomic.
- **Idea:** Such dependencies are due to InitRd's and FnlWr's.
- **Theorem:** A set T of transactions is atomic iff
  - The two conditions in the previous theorem hold, and
  - Every feasible interleaving of events in $\bigcup_{t \text{ in } T}$ InitRd(t) $\cup$ FnlWr(t) is atomic.

Experimental Results 1

Programs	Lines	Base time	On-line reduct.		Off-line reduct.		Block-based	
			Time	Report	Time	Report	Time	Report
Elevator	528	0.2s	0.2s	0-2-0-0	0.5s	0-2-0	0.6s	0-2-0
Tsp	706	0.24s	0.5s	0-0-1-0	0.5s	0-0-1	0.5s	0-0-0
Sor	251	0.47s	2.2s	0-0-2-0	53.3s	0-0-0	1.1m	0-0-0
Hedc	2197	0.6s	0.7s	0-0-2-0	1.0s	0-0-1	2.1s	0-0-0
Moldyn	1265	44.0s	9.8m	0-0-0-2	26m	0-0-0	29m	0-0-0
Montecarlo	3619	15.8s	1.7m	0-0-0-0	8.2m	0-0-0	8.2m	0-0-0
Raytracer	1832	14.3s	35m	1-0-0-1	12m	2-0-0	36m	2-0-0

Report categories: bug - benign - false_alarm - missed_violation

Experimental Results 2

Programs	Lines	Base time	On-line reduct.		Off-line reduct.		Block-based	
			Time	Report	Time	Report	Time	Report
Jigsaw	25K	1.6s	1.9s	1-2-1-1	2.7s	1-3-1	8.4m	1-3-0
StringBuffer	1255	-	-	0-1-0-0	-	0-1-0	-	0-1-0
Vector	1020	-	-	5-1-0-12	-	5-3-10	-	5-3-0
Hashtable	1054	-	-	0-2-3-0	-	0-2-3	-	0-2-0
Stack	119	-	-	4-1-0-14	-	4-3-12	-	4-3-0

Report categories: bug - benign - false_alarm - missed_violation

Summary of Experimental Results: Accuracy

- Evaluated on 12 benchmarks totaling 39 KLOC.
- **Heuristic:** public or synchronized methods should be atomic. Count the number of them that get warnings.
- **Block-based**
  - 12 bugs, 13 benign violations, no false alarms
- **Off-line reduction-based**
  - 12 bugs, 13 benign violations, 28 false alarms
- **On-line reduction-based**
  - 11 bugs, 9 benign violations, 9 false alarms
  - Missed 1 bug and several benign violations
- **Sample Bug:** Error in Vector, described earlier

Summary of Exper. Results: Performance

- **Slowdown:** running time with anal / original running time
- **Median Slowdowns:**
  - On-line reduction-based: 3 (ignoring arrays)
  - Off-line reduction-based: 17
  - Block-based: 35

Run-time Atomicity Checking: Related Work

- Atomizer [Flanagan & Freund 2004]
  - on-line reduction-based algorithm, as in our experiment
- View consistency [Artho, Havelund, & Biere 2003]
  - analyze set of vars accessed in each synchronized block
- Stale value errors [Burrows & Leino 2002, Artho+ 2004]
  - acq(l) tmp=v rel(l) v=e acq(l) result=f(tmp) rel(l)
  - Stale value errors can cause atomicity violations
  - Burrows+: static analysis. Artho+: run-time analysis.
- Run-time refinement checking [Tasiran & Qadeer 2004]
  - more general property; find actual, not potential, viol's.

Outline

- Run-time Atomicity Checking
  - More scalable.
  - Does not guarantee atomicity in other runs.
- Atomicity Types (and Race-Free Types)
  - Guarantee atomicity in all runs of the program
  - Focus on type inference, for automatic analysis
- Optimized Run-time Atomicity Checking using Types

Atomicity Types [Flanagan & Qadeer 2003]

- Associate an **atomicity type** with each code block:
  - **const:** accesses only read-only variables
  - **mover:** left-commutes and right-commutes
  - **atomic:** atomic
  - **cmpd:** compound (not atomic)
  - **error:** violates indicated locking discipline
  - **l?a:b (conditional atomicity):** atomicity is a if lock l is held, and is b otherwise.
- **Example:** If atomicity type of s is mover, then atomicity type of synchronized (l) { s } is l ? mover : atomic. Commutativity pattern is M M M or R M L, respectively.

Race-Free Types

- How to tell which accesses are movers?
- **Recall:** Race-free accesses are movers.
- Race-free types to the rescue!
- Each field declaration optionally has a guarded_by o or write_guarded_by o clause, indicating its owner (synchronization discipline) o.
  - Accesses to those fields are race-free (hence movers).
- We use our own race-free type system [Sasturkar, Agarwal, Wang, & Stoller 2005], generalizing work by [Flanagan, Abadi, & Freund, 1999-2000] and [Boyapati & Rinard, 2001].

Owners in Race-Free Types

Owner (i.e., synchronization discipline) o may be:

- a final expression: a lock that must be held
- self: the field is protected by the object's own lock
- thisThread: the field is unshared; no lock needed
- unique: there is a unique reference to the object; no lock needed
- readonly: the field is readonly; no lock needed
- a formal owner parameter of the enclosing class C
  - allows different owners for different instances of C

Example: Race-Free Types, Atomicity Types

```
class Acct<thisOwner> implements Runnable {
  int balance guarded_by thisOwner;
  Acct(Acct<unique> this,  int bal) atomicity mover {
     this.balance = bal; }
  void deposit(int x) atomicity thisOwner ? mover : error {
     this.balance = this.balance + x; }
  void run(Acct<self> this) atomicity atomic {
     synchronized (this) { this.deposit(10); } }

Acct<thisThread> a1 = new Acct<thisThread>(0);
a1.deposit(10);
Acct<self> a2 = new Acct<self>(0);
(new Thread(a2)).run();  (new Thread(a2)).run();
```

Type Inference

- Who wants to write all those type annotations?
- **Bad news:** Inference of race-free types is NP-hard.
- **What can we do?**
  - Translate to SAT; use a SAT solver [Flanagan & Freund 2004]
    - SAT solvers are impressive, but it's still NP-hard.
  - Type discovery [Agarwal, Sasturkar, & Stoller 2004]
- **Good news:** Given race-free types, inference of atomicity types is fairly easy.
  - [Flanagan, Freund, & Lifshin 2004]
  - [Sasturkar, Agarwal, Wang, & Stoller 2004]

Type Discovery

- Novel combination of static and run-time analysis
  - Run the program, guess race-free types based on the observed behavior, check them with the type checker.
- Inexpensive
- Necessarily incomplete (fails for some typable programs)
- Effective: gets 98% of annotations in our experiments
- Test suite with low branch coverage is fine!
  - Static intra-procedural type inference efficiently propagates discovered types into unexecuted branches.
- Does not discover types for unexecuted methods.
  - Can still show that parts of the program are race-free.

Type Discovery for Race-Free Types

1. Identify unique references using static analysis
2. Instrument the program to record, for each declaration d of a field, parameter, or return type, a set S(d) of objects stored there, and for each field f of each o in S(d),
   - lockset for o.f
   - whether o.f is readonly (not written after initialization)
   - whether o.f is shared (accessed by multiple threads)
   - values val(e) of final expressions e in scope at d
     - If val(e) is in lockset(o.f), then e is a candidate owner (protecting lock) of o.f.
3. Analyze log to get race-free types for these declarations.
4. Infer remaining types using intra-procedural static anal.

Type Inference for Atomicity Types

- Partial order on atomicities:
  - const ≤ mover ≤ atomic ≤ cmpd ≤ error
  - Extend pointwise to conditional atomicities
- For each method m, construct a **transfer function** f(m) that computes the atomicity of m from the atomicities of methods called by m.
- An assignment a of atomicities to methods is **consistent** if: for all m, a(m) ≥ f(m)(a(m1), a(m2), ...), where m1,m2,... are the methods called by m.
- The desired typing is the smallest consistent assignment of atomicities. Compute it with as a least fixed-point using a worklist algorithm.

Experimental Results: Race-Free Types

- Evaluated type discovery for race-free types on 11 benchmarks totaling 15 KLOC: multithreaded servers, parallel graph algs, scientific computing, web crawler, etc.
- Run-time overhead is about 20% (thanks to sampling).
- Type discovery got 98% of the annotations correct.
  - We fixed 0.9 annot/KLOC on average for all benchmarks, except one with complex synchronization that required 3 annot/KLOC.
- Found 7 bugs, 7 benign races, 24 false alarms
  - Count fields on which races are reported
  - Added final modifier to some field declarations

Experimental Results: Atomicity Types

- Inferred atomicity types for 6 benchmarks, 14 KLOC.
- Found 3 bugs, 4 benign violations, 23 false alarms
  - Count clusters containing methods reported as not atomic at some call site
  - Many false alarms are due to imprecision of race-free types for static fields and start-join synchronization.
- Showed 91% of the methods (640 out of 701) are atomic.
- **Sample Bug:** in hedc web crawler, in Worker.run():
  - Task t=handOffQueue.get(); if (t.valid()) { t.run(); }
  - t can be cancelled between the calls to t.valid and t.run
  - Run-time checking misses this (handOffQueue unused).

Expressiveness of Race-Free Type System

Program	LOC	#fields	false alarm o,m	#bugs	#ben. race
game	87	3	0,0	0	0
chat	308	7	0,0	0	0
phone	302	11	0,0	0	0
stock quote	242	6	0,0	0	0
http	563	19	0,0	0	0
elevator	523	21	1,1	0	0
tsp	706	36	14,4	4	3
hedc	7072	206	31,10	2	3
jgfutil	376	10	0,0	0	0
Barrier classes	134	3	2,0	0	1
moldyn	730	91	7,5	0	0
raytracer	1308	61	2,1	1	0
montecarlo	3198	94	26,3	0	0
Total	15549	568	83,24	7	7

Efficacy of Discovery of Race-Free Types

Program	LOC	# annotations	#annot changed
game	87	24	0
chat	308	51	0
phone	302	55	0
stock quote	242	50	0
http	563	127	0
elevator	523	56	0
tsp	706	61	0
hedc	7072	503	21
jgfutil	376	27	0
Barrier classes	134	3	1
moldyn	730	64	0
raytracer	1308	263	7
montecarlo	3198	230	0
Total	15549	1514	29

Experimental Results for Atomicity Types

Programs	Lines	Methods	False Alarms	Benign Violations	Bugs
elevator	535	24	4	0	0
tsp	736	24	1	1	0
moldyn	730	23	3	0	0
raytracer	1308	72	2	0	1
montecarlo	3198	179	4	0	0
hedc	7072	379	9	3	2
Total	13581	701	23	4	3

Static Analysis of Atomicity: Related Work

- Atomicity types [Flanagan & Qadeer 2003]
  - Related work on inference of race-free types and atomicity types was discussed earlier
  - Extended with purity [Flanagan, Freund, Qadeer 2004]
- Method consistency [von Praun & Gross 2003]
  - Based on Artho et al.'s view consistency
  - Approximate whole-program analysis
    - Under-reporting and over-reporting are possible
- Stale value errors [Burrows & Leino 2002]
  - Stale value errors can cause atomicity violations
  - Simple, efficient static analysis

Outline

- Run-time Atomicity Checking
  - More scalable.
  - Does not guarantee atomicity in other runs.
- Atomicity Types (and Race-Free Types)
  - Guarantee atomicity in all runs of the program
  - Focus on type inference, for automatic analysis
- Optimized Run-time Atomicity Checking using Types

Optimized Run-time Atomicity Checking Using Types

- Optimize the reduction-based algorithms using results from type discovery and type inference.
  - Do not monitor fields verified to be race-free by the type-checker.
  - Do not check atomicity of methods verified to be atomic by the type-checker.
- Median Slowdowns:
  - On-line reduction-based: reduce from 16 to 1.5
  - Off-line reduction-based: reduce from 38 to 2.1
- Block-based algorithm can also be optimized using types.

Atomicity of Non-Blocking Algorithms

- Above techniques are based on locks and race-freedom.
- Typical non-blocking algorithm:
  - Read values of shared variables into local variables
  - Compute on local copy
  - If no conflicting update occurred, then commit the results (write to shared variables), otherwise retry.
  - No mutual exclusion! Races are everywhere!
- Static analysis of atomicity of non-blocking algorithms [Wang & Stoller 2005]. Build on purity [Flanagan, Freund & Qadeer 2004].
  - failed attempts (which lead to retry) can be ignored

Thank you.

Any questions?

Type Discovery for Race-Free Types

1. Identify unique references using static analysis
2. Instrument the program to record (i) the set S(d) of objects stored in each field, method parameter, and method return d, and (ii) for each d and each o in S(d),

   lockSet(d,o,f): set of locks held when o.f is accessed

   rdOnly(d,o,f): whether o.f was written after initialization

   shar(d,o,f): whether o.f is accessed by multiple threads

   val(o,e), for e in FE(d): value of e for o (e could be this.f)

   FE(d): final expressions in scope at declaration of d
3. Analyze the log to discover owners of fields, method parameters, and method return values.
4. Infer remaining owners using intra-procedural static anal.

Details of Step 3: Discover Owners from Log

owner(d,f): owner of field f of objects stored in d.

The first applicable rule wins.

If Java type of d is immutable (e.g., String), then owner(d,f)=readonly

If (∀ o in S(d) : ¬shar(d,o,f)), then owner(d,f) = thisThread

If (∀ o in S(d): rdOnly(d,o,f)), then owner(d,f) = readonly

If (∀ o in S(d): o in lockSet(d,o,f)), then owner(d,f) = self

If for some e in FE(d), (∀o in S(d): val(o,e) in lockSet(d,o,f)), then owner(d,f) = e

Otherwise, owner(d,f)=fOwner, where fOwner is an owner parameter of the class containing d.