Runtime Atomicity Analysis of Multi-threaded Programs

Focus is on the paper:

“Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs”
by C. Flanagan and S. Freund

presented by Sebastian Burckhardt
University of Pennsylvania
CIS 700 – Runtime Verification Seminar
Wednesday, October 20, 2004

Outline of talk

- Verification of multithreaded programs in general
- Atomizer: the core concepts
  - Dynamic analysis
  - Reduction
  - Lock set algorithm
- Atomizer: the improvements
- Atomizer: evaluation

Correctness of Multithreaded Programs

- “Multithreaded” means: concurrent, communication by shared memory
- Reasoning quite challenging even for experts
- Typically, programmers use fairly low-level synchronization primitives
  - Mutex, Locks
  - Semaphores
  - Monitors (re-popularized by Java)
- To make it worse, performance matters (otherwise, why bother with multiple threads?)

Non-dynamic verification

- We won’t talk about these today.
  - Restrict design space
    - type systems
    - special-purpose languages
    - Design paradigms
  - Static analysis
    - Lexical
    - Control flow
    - Data flow

Checking concurrent executions

- Problem: number of possible concurrent executions very large
- Approach I: Check them all
  - means: model check the concurrent model
  - not practical without heavy abstraction
- Approach II: Check just one
  - this is the regular “testing” method
- Approach III: Check one, and extrapolate
  - look for bad things that “could” happen

What are the bad things we can look for?

- Deadlock
- Races
  - Definition of “race”: Two threads are allowed to access same variable at the same time, and at least one access is a write
- View inconsistency
  - intuitive description: grouping of variables inconsistent among threads
- <u>Lack of atomicity</u>

What are we looking for?

- Deadlock
  - look for inconsistent order of lock acquisition
- Races
  - look for variables that aren’t consistently protected by some lock, by tracking locks held during each access (e.g. “Eraser” Lockset alg)
- View inconsistency
  - track variable sets associated which each lock (e.g. in JPaX, JNuke)
- Atomicity
  - Reduction-based (e.g. Atomizer)
  - Block based (e.g. Wang/Stoller’s tool)

Atomicity Checking: Advantages

- Can find bugs that are resistant to regular testing, and race detection
- Good correspondence with programming methodology
  - easy to understand the idea
  - can verify interfaces, encouraging code reuse
  - programmer can gain confidence in code by validating atomicity assumptions
- Scalable
  - has been applied to >100k lines of Java code

Example: java.lang.StringBuffer

```
public final class StringBuffer {

  public synchronized StringBuffer append(StringBuffer sb) {
    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(0, len, value, count);
    ...
  }

  public synchronized int length() { ... }
  public synchronized void getChars(...) { ... }
}
```

Example: java.lang.StringBuffer

```
public final class StringBuffer {

  public synchronized StringBuffer append(StringBuffer sb) {
    int len = sb.length();
    ... // another thread can modify sb here
    ... // => len is no longer the correct length of len
    ... // but there is no race.
    sb.getChars(0, len, value, count);
    ...
  }

  public synchronized int length() { ... }
  public synchronized void getChars(...) { ... }
}
```

Definition

- A block of code is ‘atomic’ if for every legal execution of the program, there is an equivalent legal execution within which the entire block executes without preemption.
- Executions are “equivalent” iff
  - the (dynamic) instruction stream per thread is identical
  - the same read reads the value of the same write

How does it work? (1)

- Identify blocks that are supposed to be atomic
  - use heuristics
    - exported methods
    - synchronized methods
    - synchronized blocks
  - allow user annotations
    - can ‘turn off’ the checking if there are false bugs
    - can do additional checks by declaring atomic

```
/*# atomic */ void getChars() { ... }
```

How does it work? (2)

- Perform instrumentation on the source code level
  - could also be done at the bytecode level
  - Instrumented source code produces event stream during execution

- Analyze event stream on-line (Atomizer) or off-line.
  - For each block that is supposed to be atomic, check whether there is an equivalent execution in which it is scheduled contiguously.

How does it work? (3)

- We can’t possibly check all possible executions to find an equivalent atomic one
- Idea: Find a large class of instruction sequences for which we can always guarantee that it can be shuffled into an uninterrupted sequence by local, pairwise swaps.
- Then, warn user if supposedly atomic block does not belong to this class
- -> Lipton’s theory of reduction (1975)

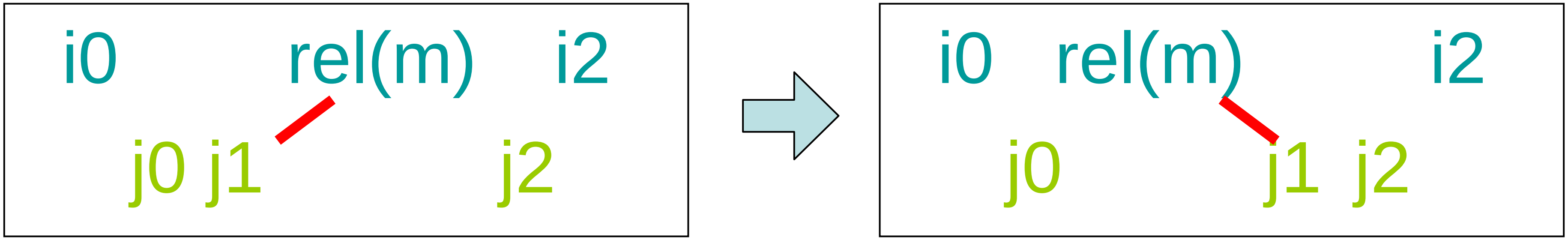
Semantic model

- Dynamic instruction stream of each thread consists of 4 types of instructions:

– rd(x,v)	read value v from shared var x
– wr(x,v)	write value v to shared var x
– acq(m)	acquire lock m
– rel(m)	release lock m

Left-movers

- Can always swap an rel(m) with an interleaved instruction j1 of another thread to its left. Call this a left-mover.



i0 rel(m) i2

j0 j1 j2

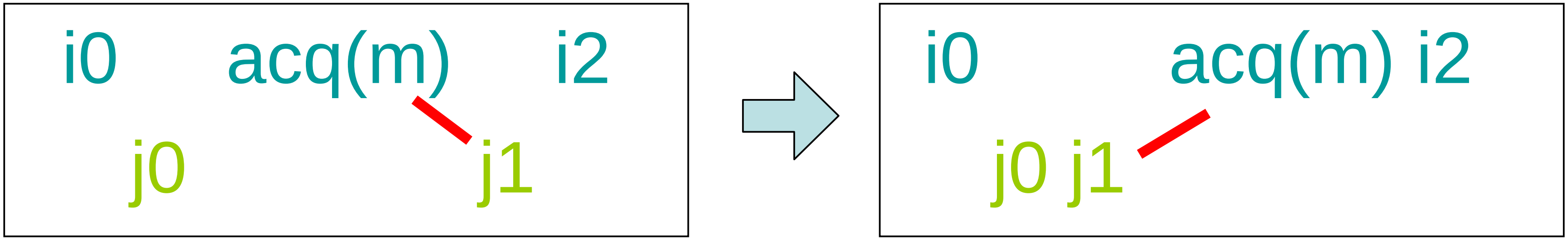
i0 rel(m) i2

j0 j1 j2

- Reason
  - – can always release lock earlier
  - – read/write matching not affected by move

Right-movers

- Can always swap an acq(m) with an interleaved instruction j1 of another thread to its right. Call this a right-mover.



i0 acq(m) i2

j0 j1

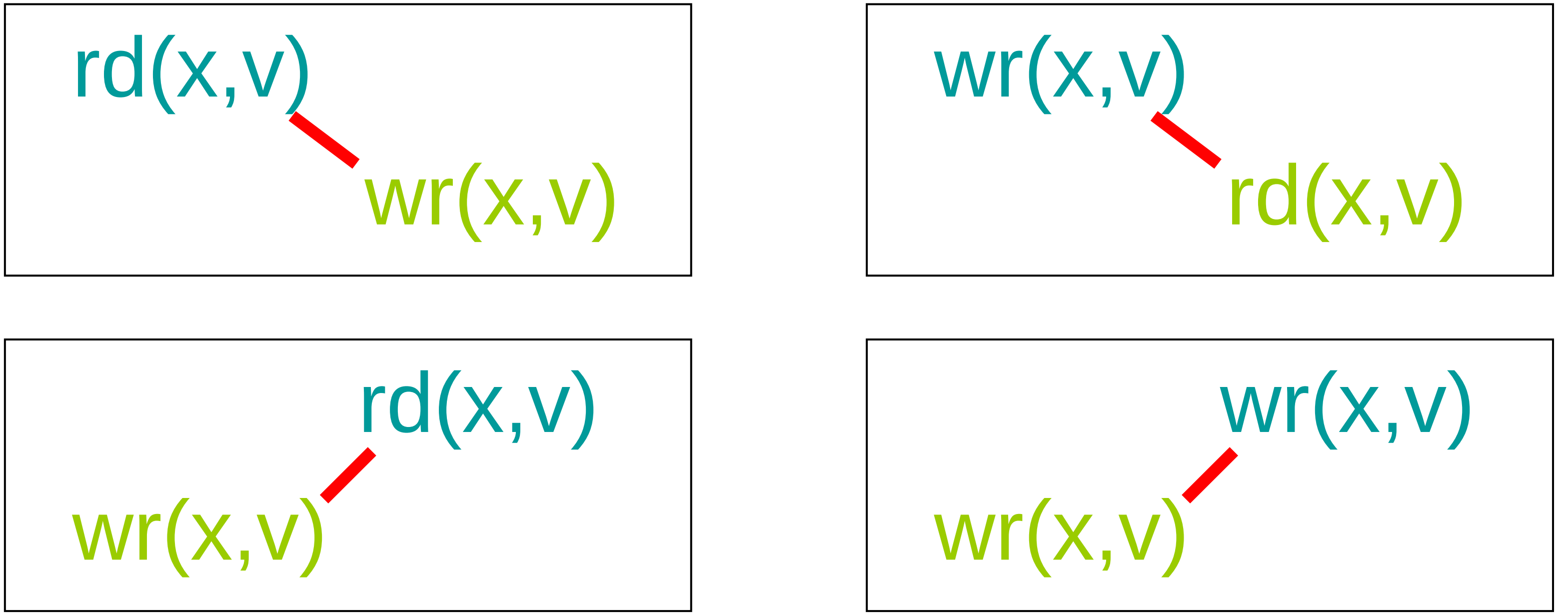
i0 acq(m) i2

j0 j1

- Reason
  - – lock is still available (j1 can not be acq(m))
  - – read/write matching not affected by move

Non-movers

- Neither rd(x,v) nor wr(x,v) can in general be swapped with an adjacent interleaved instruction of another thread. Call them non-movers.



rd(x,v)
wr(x,v)

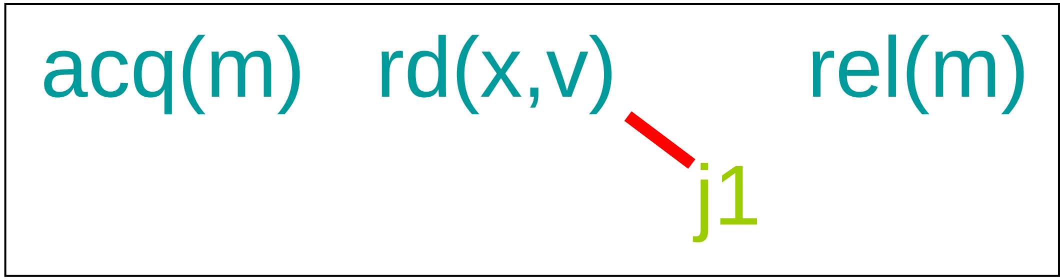
wr(x,v)
rd(x,v)

rd(x,v)
wr(x,v)

wr(x,v)
wr(x,v)

Both-movers

- If an access rd(x,v),wr(x,v) goes to a variable protected by a lock which is held by this thread, it is a both-mover.



- Reason
  - j1 can not be an access to x
- Suppose for now we know which locks can protect a variable

Lipton’s Reduction

- Let’s denote the instructions as follows: L for left-mover, R for right-mover, N for non-mover, B for both-mover
- Then any execution sequence matching the following regular expression is equivalent to an atomic one:

$$(R + B)^* \ (N + \varepsilon) \ (L + B)^*$$

- Examples: RL RBL NLLLB RNL BBB
- But not: NN LR

Example

- Say the method “copy()”

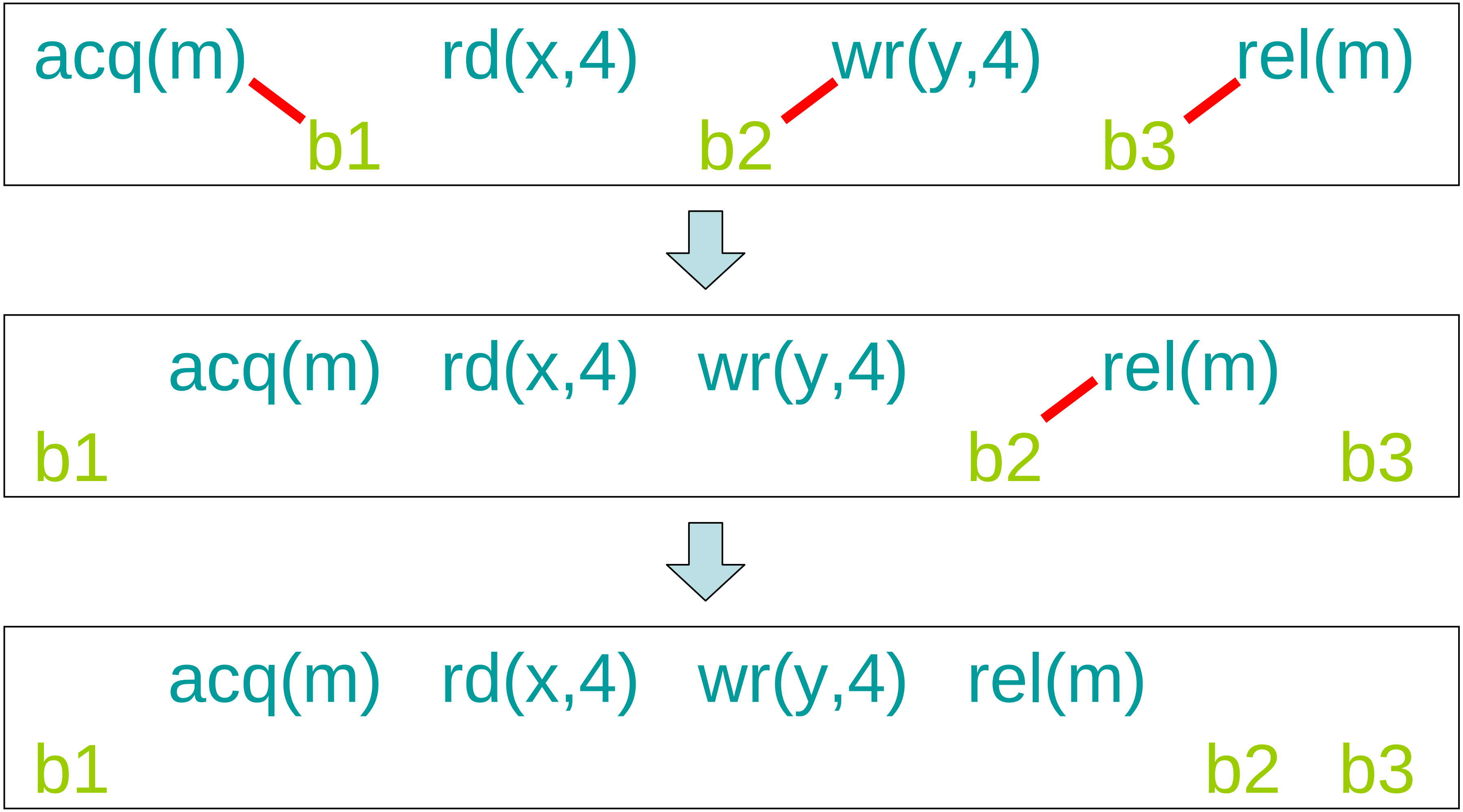
```
public class A {
  private int x, y;
  public synchronized void copy() {
    y = x;
  }
  ... (more methods) ...
}
```

- produces the dynamic instruction stream

acq(m)   rd(x,4)   wr(y,4)   rel(m)

- is it atomic?
  - For now, assume all methods of class A are synchronized

Example



acq(m) rd(x,4) wr(y,4) rel(m)

b1 b2 b3

acq(m) rd(x,4) wr(y,4) rel(m)

b1 b2 b3

acq(m) rd(x,4) wr(y,4) rel(m)

b1 b2 b3

Implementation

- Can efficiently check if blocks match
  $(R + B)^* (N + \varepsilon) (L + B)^*$
  by using an online automaton.
- Problem: to classify variable accesses correctly, we need to know which locks protect which

Which locks with which field?

fields may not be protected by this object’s lock

```
public class A2 {
  private int x,y;
  public synchronized swap()  { int z = y; y = x; x = z;}
  public int getX()  { return x; }
  public int getY()  { return y; }
  ...
}
```

field may be protected by a different object’s lock

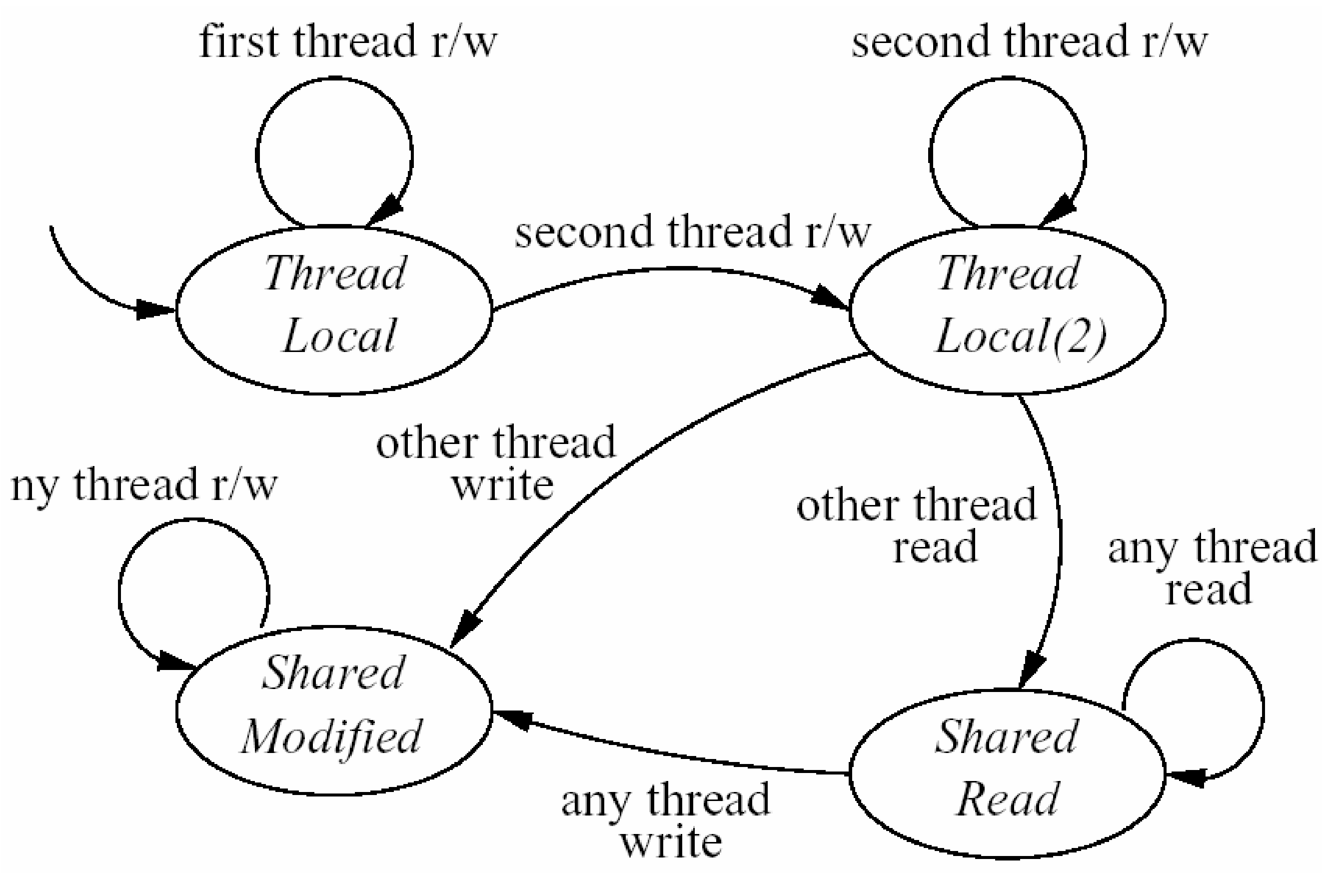
```
public class A2 {
  private int x,y;
  Integer mylock = new Integer(0);
  public copy() { synchronized(mylock) { y = x; } }
  public int getX()  { synchronized(mylock) { return x; } }
  public int getY()  { synchronized(mylock) { return x; } }
  ...
}
```

Basic “Eraser” lockset algorithm

- Argue: “If a variable is consistently protected by some lock, this lock must be held during all accesses to that variable”
- Dynamically, we can look at the set of locks helds during each access so far, and keep track of their intersection
  - If the intersection is empty, there seems to be no consistent locking discipline - classify access as a non-mover
  - Otherwise, there seems to be a consistent locking discipline - classify access as a both-mover
- What about re-classifying accesses if changes occur during runtime?
  - can’t be done on-line, but could be done off-line

Improve Classification (1)

- Avoid flagging some classic, safe usages
  - thread-local variables: need no lock to protect them
  - initialization: one thread initializes data, then passes it to another thread, thread-local from there on
  - Write once, read many times



- Track state for each field
  - use lock set for classification only if in state Shared Modified

Improve Classification (2)

- Re-entrant locks
  - re-entrant acquires and releases are both-movers
- Redundant locks
  - if a lock is only accessed by one thread, it is redundant (thread-local locks)
  - if lock B is always held while accessing lock A, lock A is redundant (protected locks)
  - redundant acquires and releases are both-movers
  - can classify locks using the same lockset and thread-access algorithms as for fields

Improve Classification (3)

- “Benign” read/write races

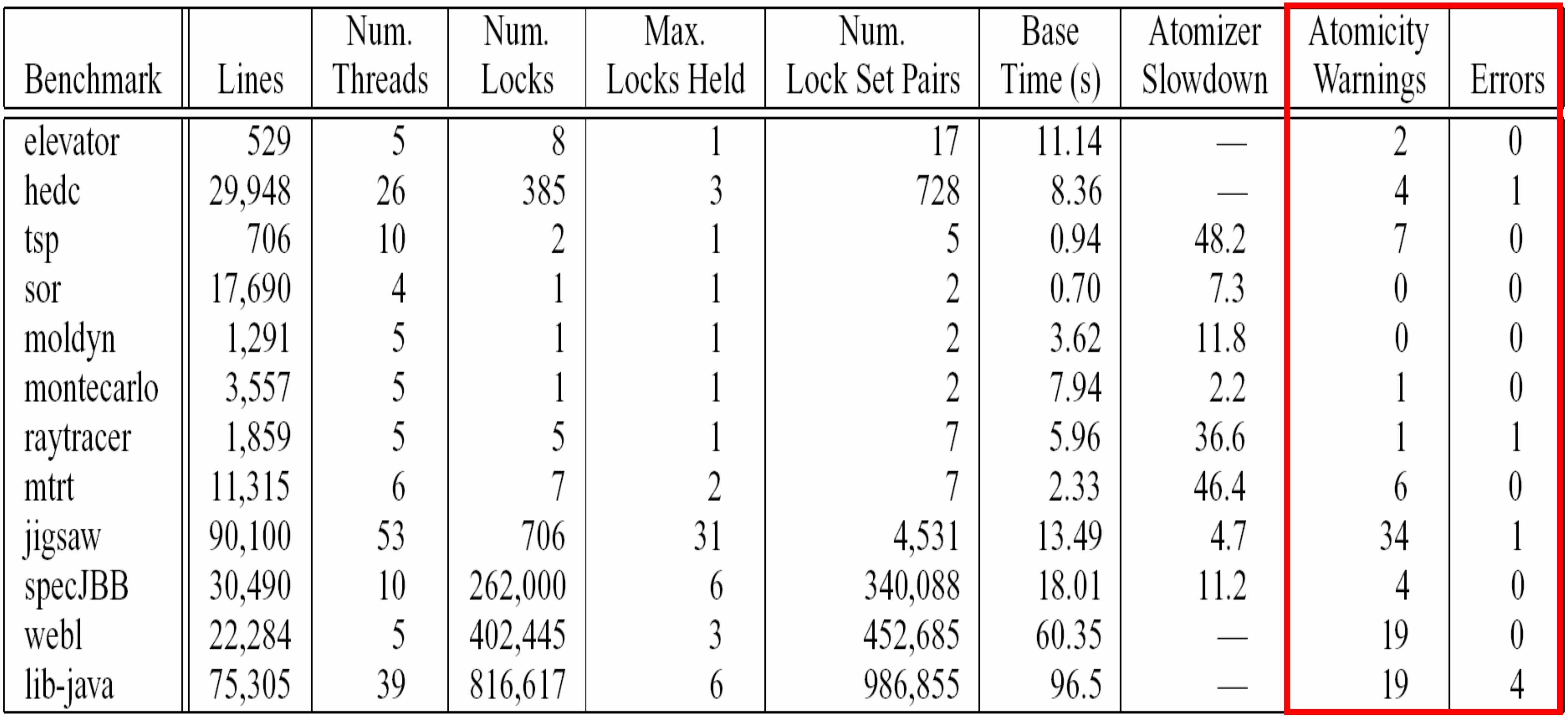
```
public class A2 {
  private int x;
  public int read()  { return x; }
  synchronized void inc() { x = x + 1; }
}
```

- read() and inc() are atomic... (more or less)
  - track separate lockset containing locks held during all writes (= superset of locks held during all accesses)
  - classify read as both-mover if current thread holds a write lock, even if access-protecting lockset is empty

It’s not that easy

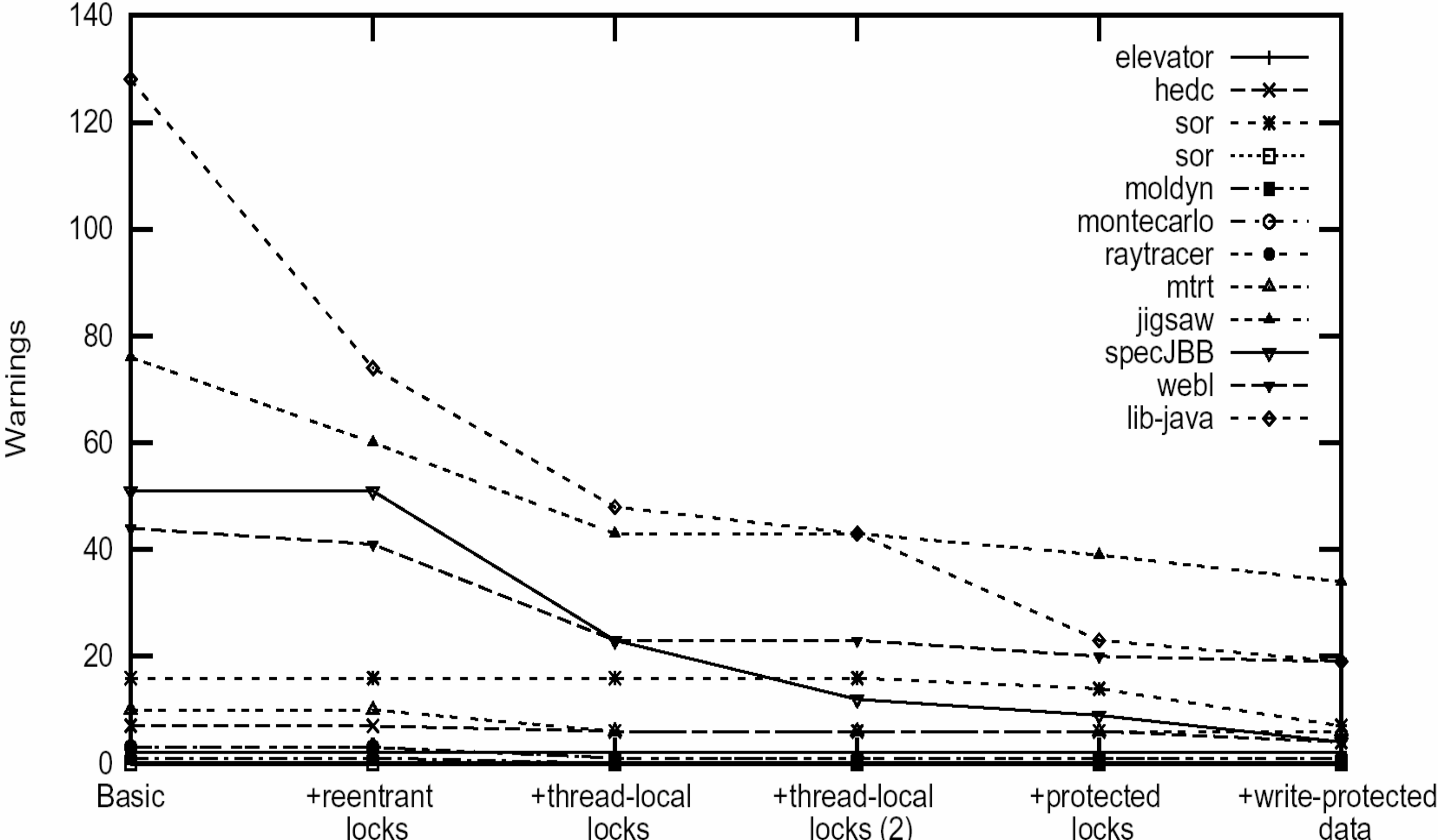
- Unsynchronized reads and writes
    - are not atomic if more than 32 bit quantity
        - more rules exists (e.g. volatile vs. non-volatile)
    - are not guaranteed to proceed in order
        - only synchronization events are sequentially consistent.
        - memory model relative to hardware is specified (?)
        - memory model of hardware is not specified.
        - does anybody know?
    - does Atomizer need adjustments for non-sequentially consistent machines?

Evaluation



Benchmark	Lines	Num. Threads	Num. Locks	Max. Locks Held	Num. Lock Set Pairs	Base Time (s)	Atomizer Slowdown	Atomicity Warnings	Errors
elevator	529	5	8	1	17	11.14	—	2	0
hedc	29,948	26	385	3	728	8.36	—	4	1
tsp	706	10	2	1	5	0.94	48.2	7	0
sor	17,690	4	1	1	2	0.70	7.3	0	0
moldyn	1,291	5	1	1	2	3.62	11.8	0	0
montecarlo	3,557	5	1	1	2	7.94	2.2	1	0
raytracer	1,859	5	5	1	7	5.96	36.6	1	1
mtrt	11,315	6	7	2	7	2.33	46.4	6	0
jigsaw	90,100	53	706	31	4,531	13.49	4.7	34	1
specJBB	30,490	10	262,000	6	340,088	18.01	11.2	4	0
webl	22,284	5	402,445	3	452,685	60.35	—	19	0
lib-java	75,305	39	816,617	6	986,855	96.5	—	19	4

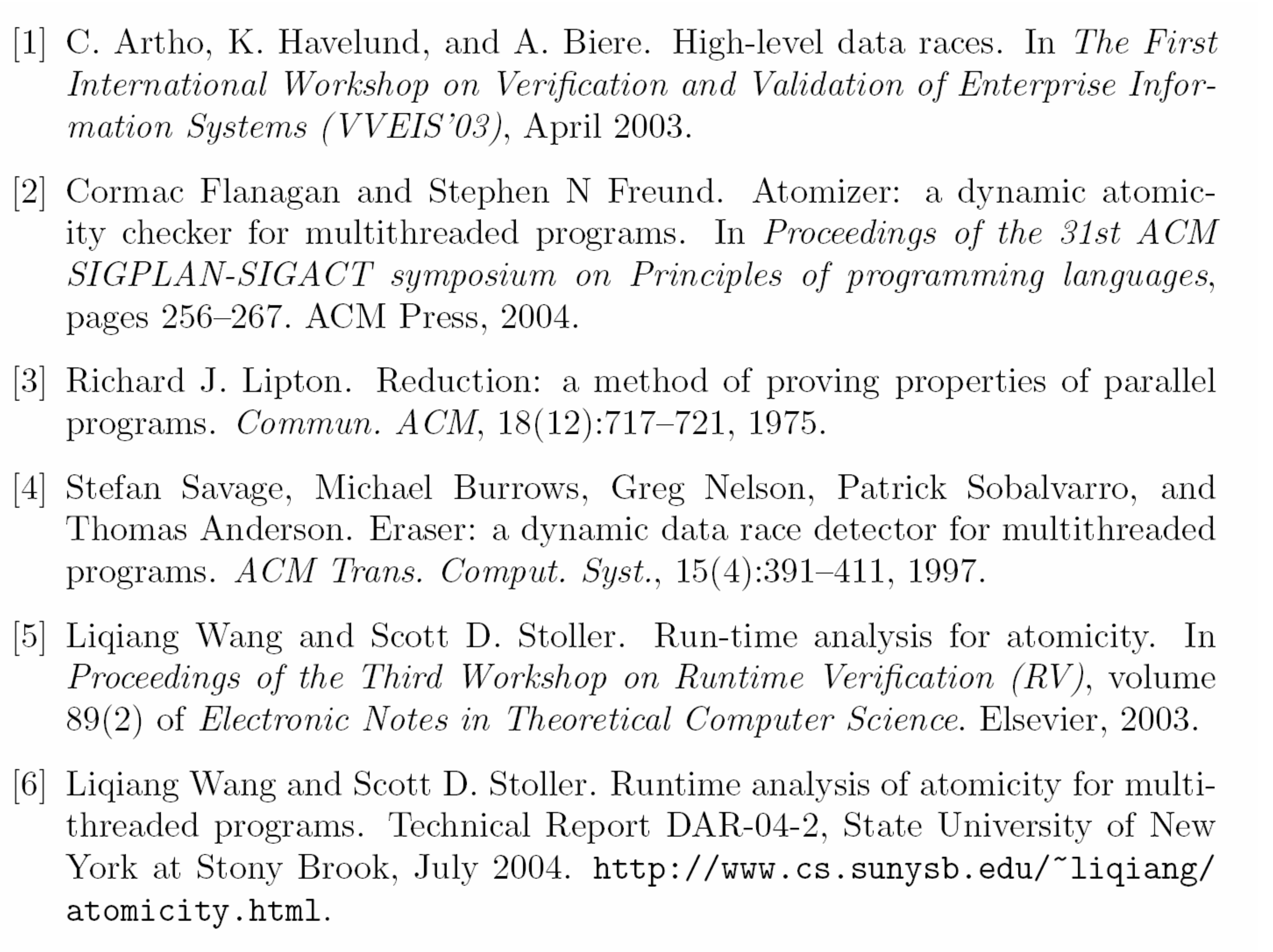
Effect of improvements



Atomizer paper: contributions

- Concise review of concepts
  - Formal semantics for multithreaded programs
  - Reduction idea, Lockset algorithm
- Description of the algorithm and some improvements
  - Formal description of the algorithm, formulation of theorem describing its correctness, in provable detail
  - Mentions optimizations: handle re-entrant locks, thread-local locks, protected Locks, write-protected data
- Experimental evaluation of the tool
  - performance, scale, usability

Bibliography



[1] C. Artho, K. Havelund, and A. Biere. High-level data races. In *The First International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, April 2003.

[2] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267. ACM Press, 2004.

[3] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[4] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[5] Liqiang Wang and Scott D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Third Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[6] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. Technical Report DAR-04-2, State University of New York at Stony Brook, July 2004. `http://www.cs.sunysb.edu/~liqiang/atomicity.html`.