Dynamic Analysis of Multithreaded Programs

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391-411, 1997.

C. Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

Motivation

- Multithreaded programming common programming technique
  - Many operating systems support threads
  - Many applications are multithreaded
- Multithreaded programming is difficult and error prone
  - Nondeterministic execution makes debugging a headache
  - Timing-dependent errors difficult to locate

Program Analysis Solutions

- Remove burden from programmer
- Static Analysis is problematic
  - Requires statically reasoning about program's semantics
  - Many techniques do are not scalable (i.e. enumerating all possible interleavings)
- Dynamic Analysis
  - Dynamic race detection (i.e. *Eraser*)
  - Dynamic atomicity checker (i.e. *Atomizer*)

Outline

- Eraser: Detecting Data Races
  - Background (Data races and previous work)
  - Improving Locking Discipline
  - Implementation and Performance
  - Experience
- Atomizer: Atomicity Checker
  - Background (Eraser and Lipton's theory of reduction)
  - Theory of Reduction
  - Implementation and Evaluation
- Conclusion

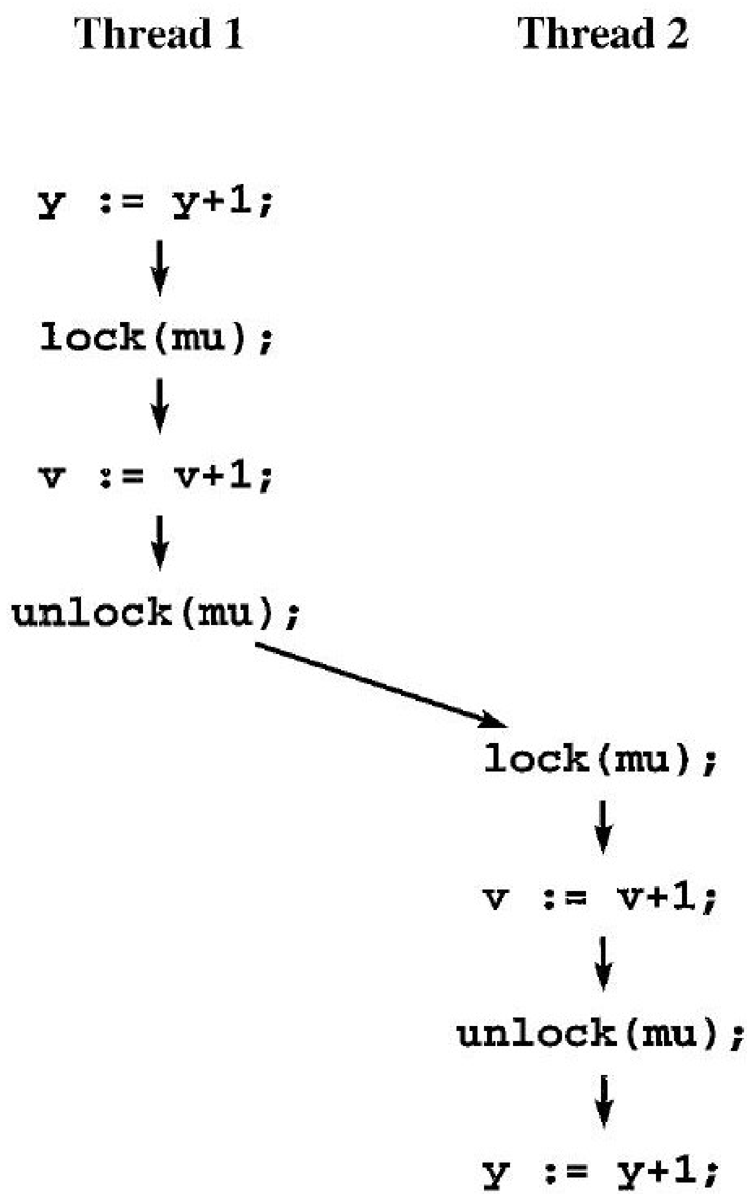
Data Race

- Lock: simple synchronization object used for mutual exclusion
  - Operations on lock mu are lock(mu) and unlock(mu)
  - Only owner of lock is allowed to release it
  - Lock is either available or owned by some thread
- Data race: occurs when two concurrent threads access a shared variable and when..
  - At least one access is a write
  - Threads use no explicit mechanism to prevent the accesses from being simultaneous

Detecting Data Races

- Lamport's *happens-before* relation
  - Partial order on all events of all threads in a concurrent execution
  - Within single thread events ordered in order that they occur
  - Between threads, events ordered according to properties of synchronization objects they access
  - If two threads access a shared variable, and the accesses are not ordered by the happens-before relation, a data race could have occurred

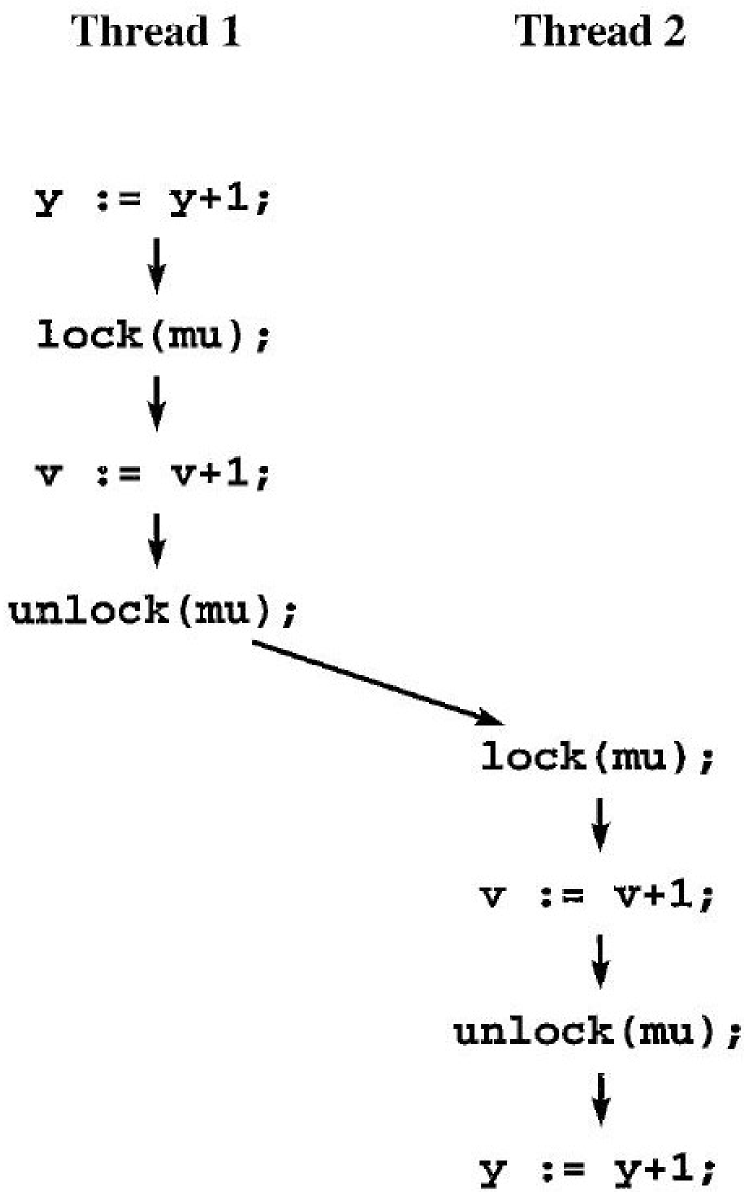
Detecting Data Races (2)

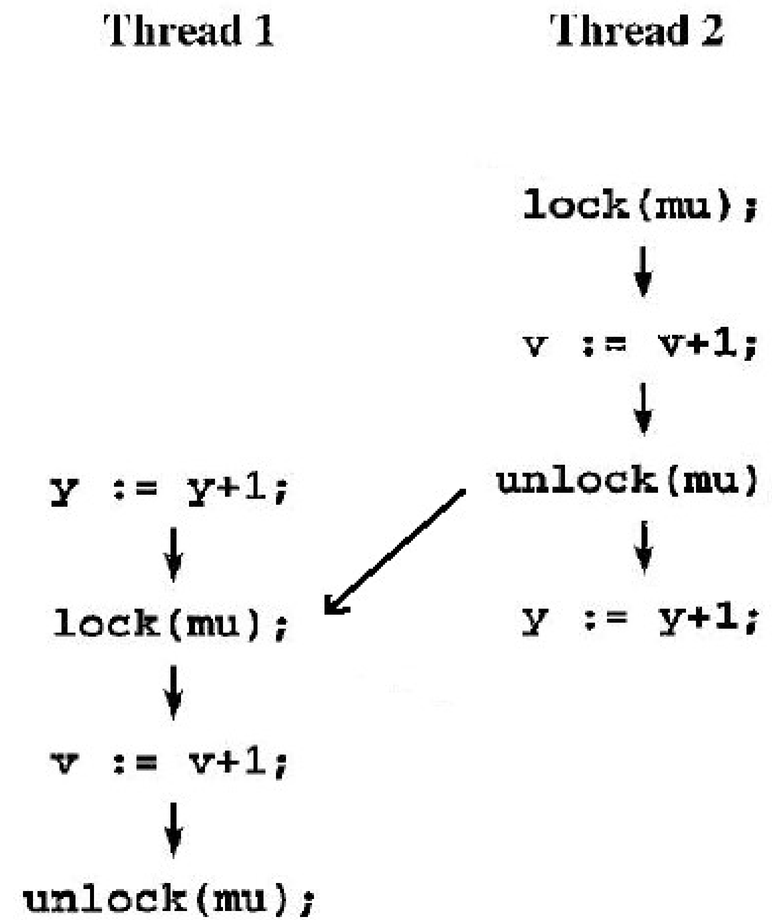


Detecting Data Races (3)

- Problems with Lamport's *happens-before relation*
  - Difficult to implement efficiently because they require per-thread information about concurrent access to each shared-memory location
  - Highly dependent on the interleaving produced by the scheduler
- Recall the previous slide

Detecting Data Races (4)





Detecting Data Races (5)

- The Lockset Algorithm
  - Every shared variable access is protected by some lock
  - Monitor all reads/writes of shared variables and make sure some lock protects the variable
  - Must infer intention of locks (*C(v)* is the set of candidate locks for variable *v*)

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
    set $C(v) := C(v) \cap locks\_held(t)$;
    if $C(v) = \{\ \}$, then issue a warning.

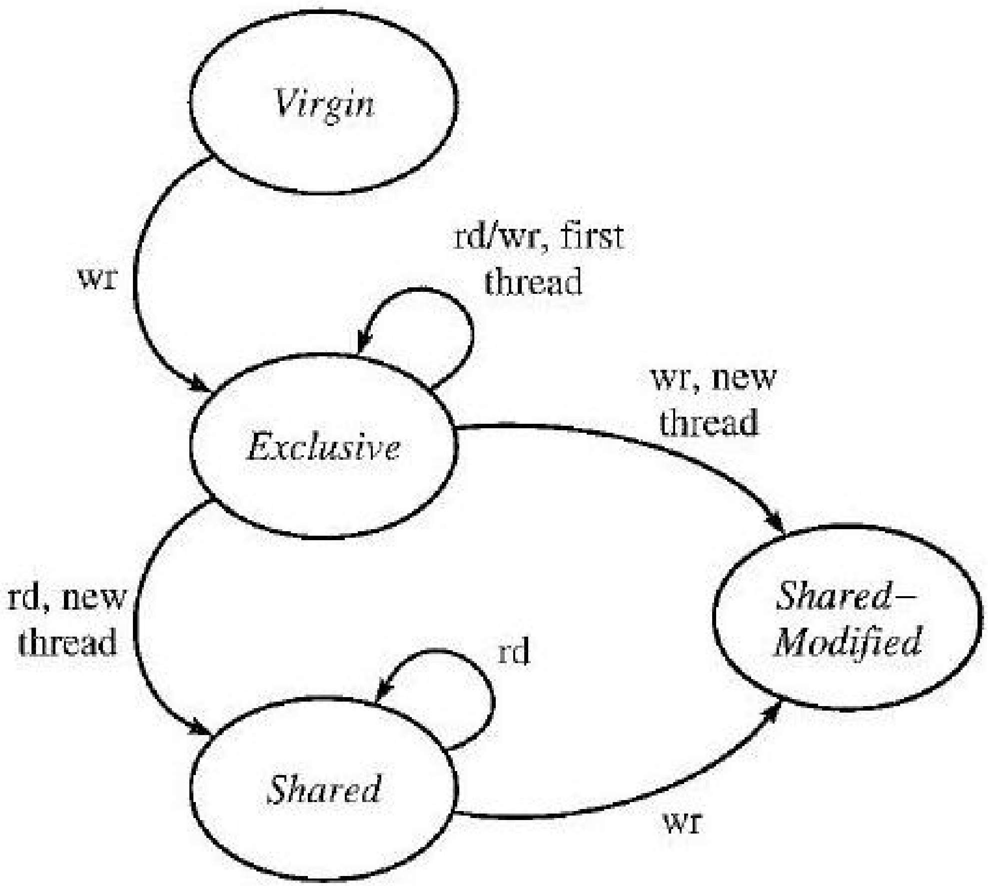
Detecting Data Races (6)

- Simple Lockset algorithm is too strict
  - Initialization: Shared variables are frequently initialized without holding any locks
  - Read-shared data: Some shared variables are written during initialization and read-only thereafter. Should allow read access without locks.
  - Read-write locks: Allow multiple readers to access a shared variable, but only a single writer
- *Eraser* extends Lockset algorithm to address these issues

Eraser: Improving Locking Discipline

- Initializing new variables
  - Delay refinement of *C(v)* until after it has been initialized
  - Consider variable initialized when it is first accessed by a second thread
  - Until then, access have no effect on *C(v)*
- Multiple reads of shared variable not races
  - No need to protect read-only variable
  - Report races only after initialized variable has become write-shared by more than one thread

Eraser: Improving Locking Discipline (2)



- Virgin: variable allocated by not referenced
- Exclusive: accessed only by one thread (do not update *C(v)*)
- Shared: read-shared data (update *C(v)* but do not report)
- Shared-modified: original rules apply

Eraser: Read-Write Locks

- Support for single-writer, multiple reader locks
  - Locks are either in write mode or read mode
  - Require for each variable *v*, some lock *m* is held in write mode for every write of *v*, and *m* is held in some mode (read or write) for every read of *v*
- Change to algorithm for Shared-modified state

Let $locks\_held(t)$ be the set of locks held in any mode by thread t.
Let $write\_locks\_held(t)$ be the set of locks held in write mode by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each read of v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) := \{ \}$, then issue a warning.
On each write of v by thread t,
  set $C(v) := C(v) \cap write\_locks\_held(t)$;
  if $C(v) = \{ \}$, then issue a warning.

Implementing Eraser

- Implemented for Digital Unix OS on the Alpha processor
- Uses ATOM binary modification system
- Instruments binary that includes calls to Eraser runtime to implement Lockset algorithm
  - To maintain *C(v)* instrument each load and store
  - To maintain *lock_held(t)* instrument each call to acquire or release a lock and thread initialization and finalization
  - To initialize *C(v)* instrument call to storage allocator

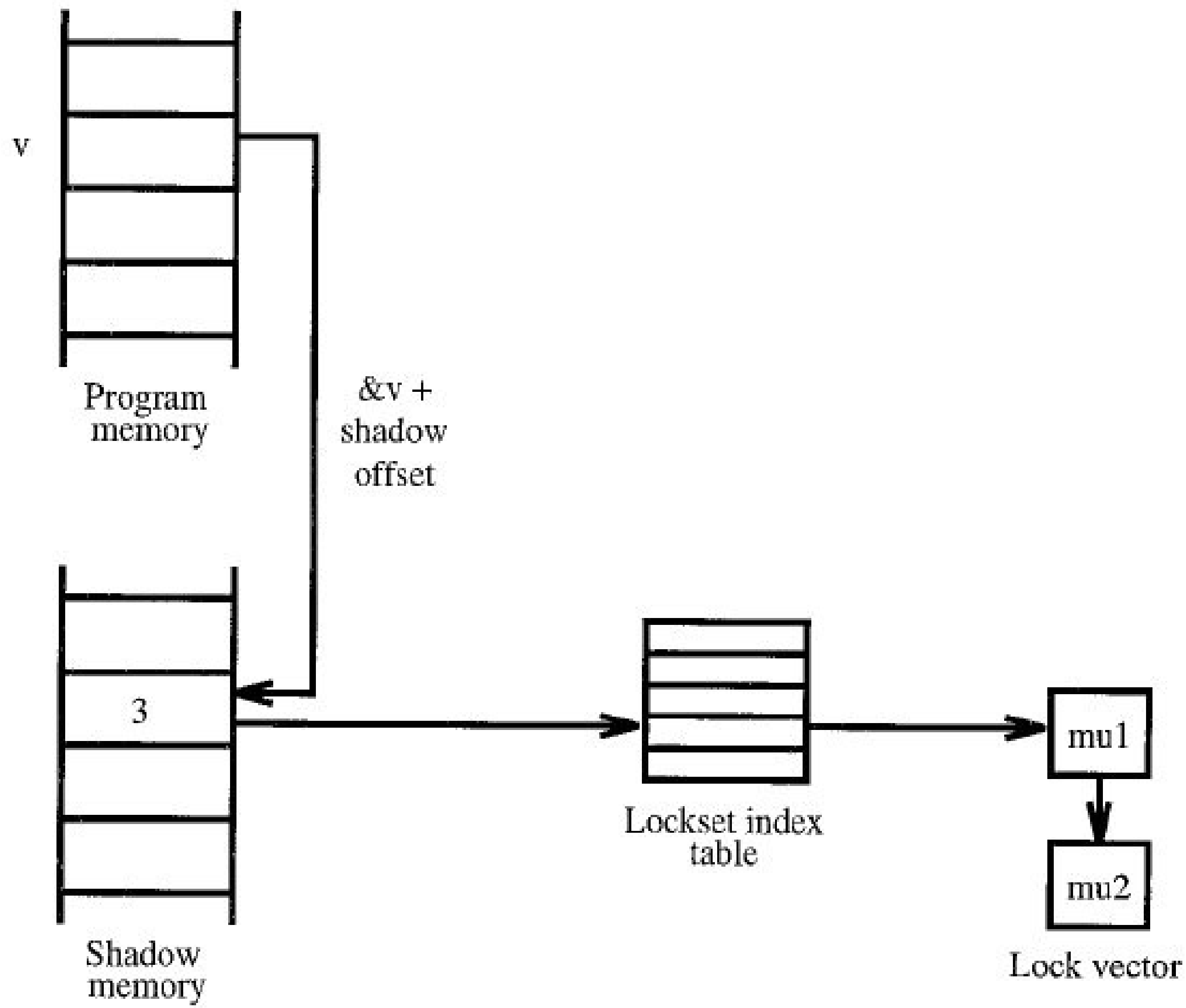
Implementing Eraser (2)

- Treat each 32-bit word in the heap or global data as a possible shared variable
  - Loads and stores whose address mode is indirect off the stack pointer are not instrumented
  - No deliberate plan to support programs that share stack locations between threads
- For each 32-bit word in the data segment and heap, keep corresponding shadow word
  - 2-bits for state condition
  - 30-bit lock set index (in Exclusive state 30 bits used to store ID of thread with exclusive access)

Implementing Eraser (3)

- Lock set index: represent set of locks by a small integer
  - Relatively small number of *distinct* lock sets
  - Only need one copy of each distinct lock set
  - New lockset indexes created as a result of lock acquisitions, lock release, or through intersection operations
  - Maintain hash table to complete lock vectors

Implementing Eraser (4)



Implementing Eraser (5)

- When race detected, Eraser indicates
  - File and line number at which it was discovered
  - Backtrace listing of all active stack frames
  - Thread ID, memory address, type of memory access, and important register values such as PC and stack pointer
- User can also direct Eraser to log all accesses to particular variable that result in a change to its candidate lock set

Implementing Eraser (6)

- Removing false alarms is key to making this tool usable and effective
- Program annotations introduced
  - Memory reuse: EraserReuse(address, size)
  - Private locks: EraserReadLock(lock), EraserReadUnlock(lock), EraserWriteLock(lock), EraserWriteUnlock(lock)
  - Benign races: EraserIgnoreOn(), EraserIgnoreOff()

Eraser Performance

- Slow down by a factor of 10 to 30 times
  - Can change order of scheduled threads, affecting behavior of time-sensitive applications
  - Could be important for very time-sensitive applications
- Procedure call at every load/store instruction
  - Could inline monitoring code
  - Could also explore opportunities for static analysis to reduce overhead of monitoring

Eraser Experience

- Large multithreaded servers written by experienced researchers at Digital Equipment Corporation's System Research Center
- Undergraduate programming assignments at University of Washington
- False alarms suppressed with annotations
  - Detected race conditions and false alarms, then modified program appropriately with annotations and reran to locate remaining problems
  - Ten iterations of this process usually sufficient to resolve all of a program's reported races

Eraser Experience: AltaVista

- Lightweight AltaVista HTTP server
  - 5000 lines of C/100 locks/250 distinct lock sets
  - Found benign data races (updates to global configuration data and statistics)
  - 24 annotations to reduce reported races to zero
- AltaVista indexing engine
  - 20,000 lines of C/900 locks/3600 distinct lock sets
  - Introduced two race conditions from project history
    - Eraser easily detected races
  - 19 annotations to reduce reported races to zero

Eraser Experience: Vesta Cache Server and Petal

- Vesta: advanced software configuration management system
  - 30,000 lines of C++/26 locks/70 different lock sets
  - Found one serious data race
  - 10 annotations for false warnings
- Petal distributed storage system: presents clients with huge virtual disk implementation by cluster of servers
  - 25,000 lines of C/64 concurrent workers
  - Found two intentional race where global variables containing statistics were modified without locking

Experience: Undergraduate Coursework

- Programs
  - Build locks from test-and-set operation
  - Build small threads package
  - Build semaphores and mutexes
  - Producer/consumer-style problems
- 100 runnable programs
  - 10% had data races found by Eraser
  - False alarm: Queue implicitly protected elements by accessing the queue through locked head and tail fields

Problem with Detecting Race Conditions

Excerpt from `java.lang.StringBuffer`

```
public final class StringBuffer {

    public synchronized
          StringBuffer append(StringBuffer sb) {
      int len = sb.length();
      ...  // other threads may change sb.length(),
      ...  // so len does not reflect the length of sb
      sb.getChars(0, len, value, count);
      ...
    }

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

Problem with Detecting Race Conditions (2)

- Absence of race conditions not sufficient to ensure absence of errors due to unexpected interference between threads
- Authors claim recent results show subtle defects of similar nature are common
  - NASA's Remote Agent spacecraft controller
  - Comparable defects in many Java applications
- Need more systematic methods for controlling interference between concurrent threads

Atomicity

- Corresponds to natural programming methodology
- Provides strong, maximal, guarantee of non-interference between threads
- Reduces challenging problem of reasoning about behavior in a multithreaded context to simpler problem of sequential behavior

Atomicity Requirement

- Serialized semantics: A thread can only perform an operation if no other thread is in an atomic block
- Standard semantics: Language implementations admit additional transitions sequences and behaviors
- Atomicity Requirement: Any correctly synchronized program execution under standard semantics should have an equivalent execution under serialized semantics

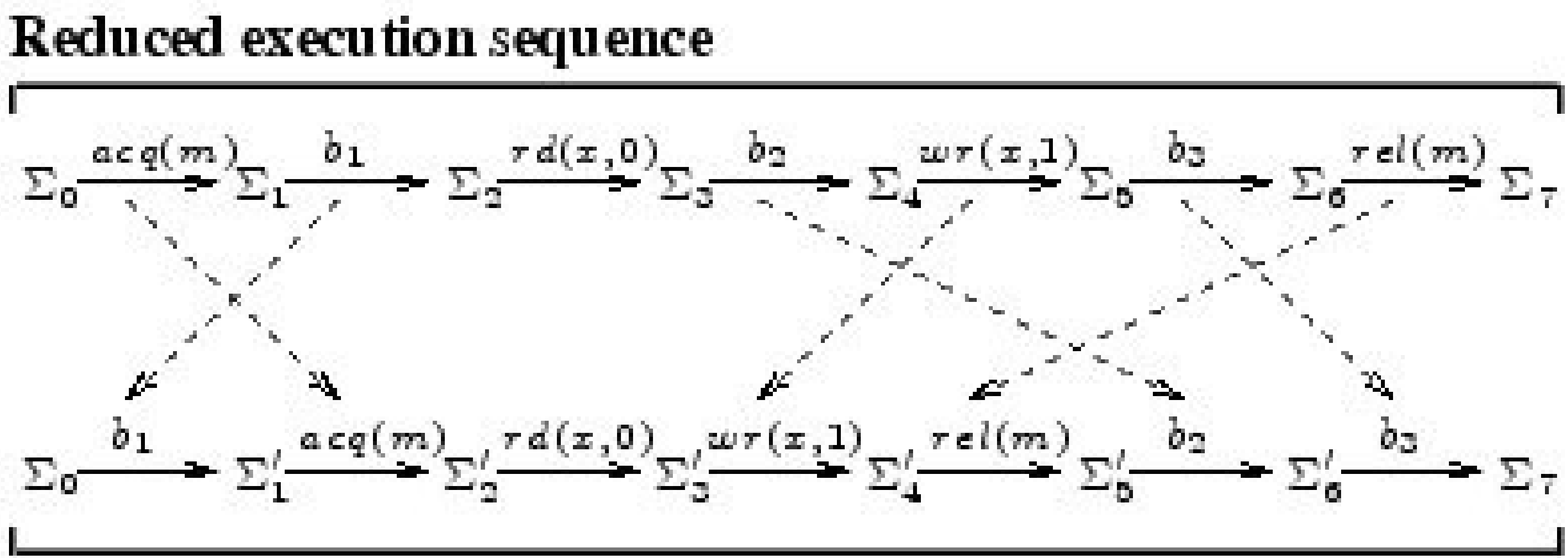
Lipton's Theory of Reduction

- Theory used to reduce execution paths under standard semantics to an equivalent serial execution
- Right-mover: An action *b*, such that for any execution the action *b* performed by one thread is immediately followed by an action *c* of a concurrent thread, the actions *b* and *c* can be swapped without changing resulting state
  - Example: acquire lock

Lipton's Theory of Reduction (2)

- Left-mover: An action *c* where *c* immediately follows an action *b* of a different thread, and the actions *b* and *c* can be swapped without changing resulting state
  - Example: release lock
- Both-movers
  - Example: protected read/write operations
- Non-movers
  - Example: unprotected read/write operations

Lipton's Theory of Reduction (3)



Lipton's Theory of Reduction (3)

- More generally: If path through a code block contains a sequence of right-movers, followed by at most one non-mover action and then a sequence of left movers, the path can be reduced to an equivalent serial execution
- Atomizer leverages theory of reduction to verify atomicity dynamically

Checking Atomicity via Reduction

- Assume we know what lock protects each variable
- Develop an instrumented semantics that only admits code paths that are reducible
- Keep track of whether thread in right-mover or left-mover part of atomic block (either *InRight* or *InLeft*)
- Every thread starts out as *InRight*

Instrumented Operations

- Operations: read, write, acquire lock, release lock, begin atomic block, end atomic block
- Protected read/write access does not change state (*InRight/InLeft*)
- Unprotected read/write access outside of atomic block: OK, state stays the same
- Unprotected read/write access in atomic block
  - If *InRight* change to *InLeft*
  - If *InLeft* --> WRONG

Instrumented Operations (2)

- Acquiring a lock:
  - In atomic block: Must be *InRight* state or else WRONG
  - Not in atomic block: OK, state stays the same
- Release a lock: state changed to *InLeft*
- Begin atomic block:
  - Already in atomic block: OK, state stays the same
  - Not in atomic block: State becomes *IsRight*
- End atomic block: OK, state stays the same

Inferring Locks

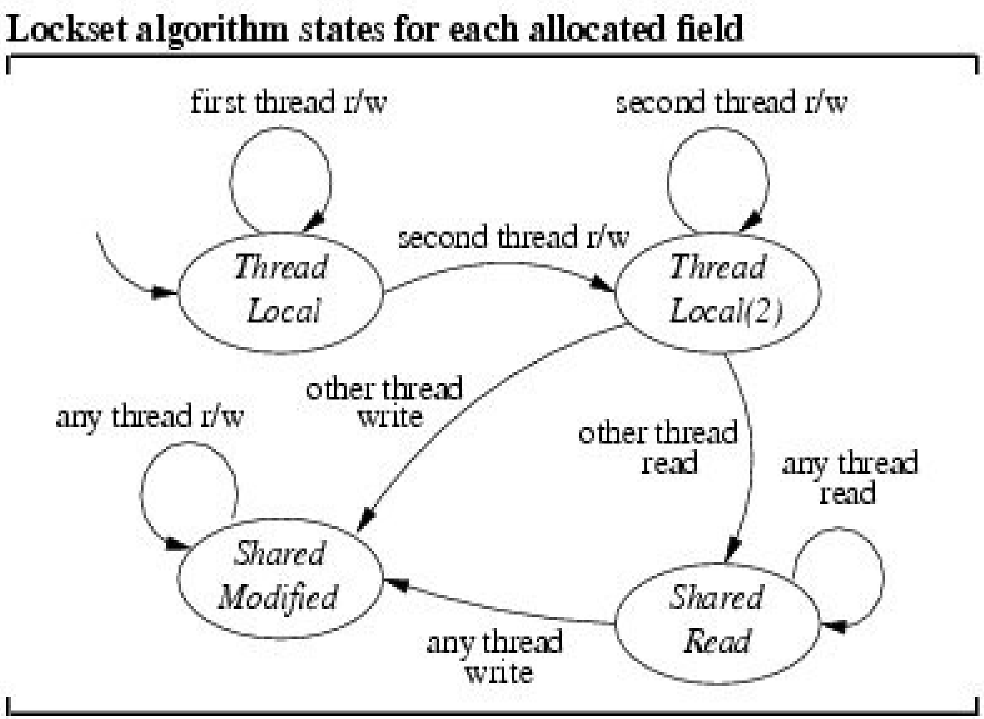
- Approach assumed knowledge of locks
- Infer protecting locks using same technique as Eraser
- If candidate lock set for variable *x* becomes empty, all accesses to that variable treated as non-movers
- Problem: Previous accesses to *x* may have been incorrectly classified as both-movers

Inferring Locks (2)

```
/*# atomic */ void double() {
  synchronized (m) {
    int t = x;
    x = 2 * t;
  }
}
```

- *x* classified as both-mover since protected by lock *m*
- Between operations another thread accesses *x* with no lock
- *x* classified as non-mover on second operation of double

Lockset algorithm



- Thread-local: only accessed by local thread
- Thread-local (2): ownership transferred to second thread (common initialization pattern in Java)
- Shared Read
- Shared Modified

Implementation

- Instruments Java source code
- Programmer-supplied annotations for atomic blocks
- Supports annotations to suppress spurious warnings, ignore races on specific fields, etc.
- Heuristics to automatically decide atomic blocks
  - All public/protected methods of a class
  - All synchronized blocks and methods

Extensions

- Eliminating false positives: classification of lock operations are sometimes overly conservative
- Extensions to Atomizer
  - Re-entrant locks: Acquire is a both-mover, since it cannot interact with other threads
  - Thread-local locks: If lock used by only a single thread, acquire and release are both-movers
  - Thread-local locks (2): Eliminates false alarms caused when one thread creates and initializes protected object and transfers ownership of the object and protecting locks to another thread

Extensions (2)

- More extensions to Atomizer
  - Protected locks: Threads always hold some lock m1 before acquiring m2, operations on m2 are both-movers
  - Reader/Writer locks (same as Eraser)
    - Read both-mover if current thread holds at least one of the write-protecting locks; otherwise non-mover
    - Write both-mover if holding some (read or write) lock; otherwise non-mover

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

Figure 1. Summary of test programs and performance.

Evaluation (2)

- Atomizer identified a number of potentially damaging errors in mature software
- Instrumented Java libraries
  - In synchronized method PrintStream.println(String s)
  - Two threads can write to variable out which could cause the output stream to be corrupted
  - Atomizer caught error with no programmer intervention and pinpointed exact location in program where bug could manifest itself

Evaluation (3)

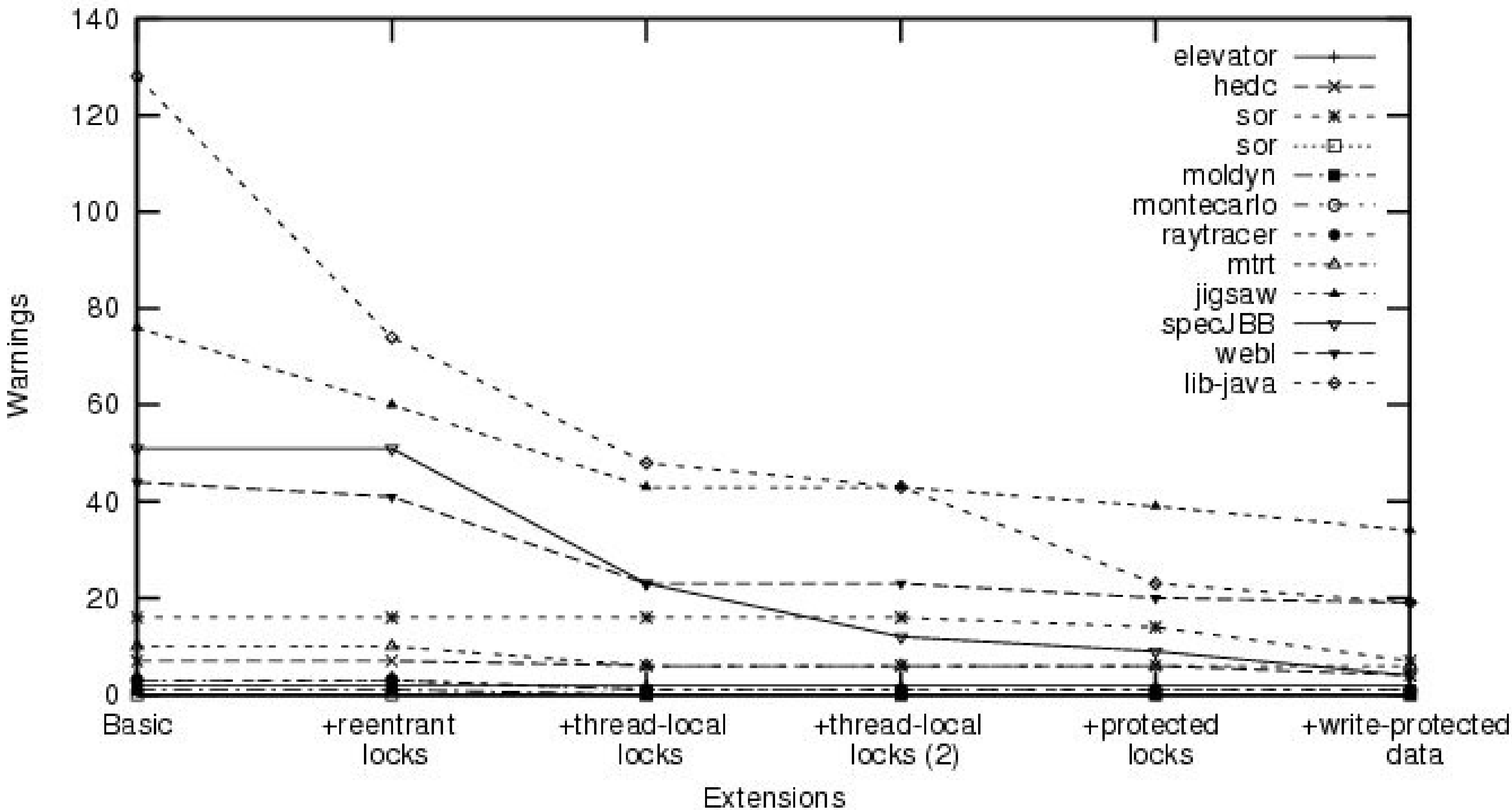


Figure 2. Warnings reported by the Atomizer under different configurations.

Conclusions

- Developing multithreaded software difficult
- Eraser: Dynamically detect race conditions
  - Extends Lockset algorithm
  - Experience shows effective
- Atomizer: Dynamically checks for atomicity
  - Removing race conditions is not enough
  - Atomicity fundamental design principle in multithreaded program
  - Uses theory of reduction to ensure atomicity requirement

Questions?