CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 9: March 8

Lecturer: Emery Berger *Scribe: Heather Conboy*

9.1 Overview

Atomicity is a correctness property of multithreaded programs that guarantees that concurrent threads do not interfere with atomic methods (and/or blocks). This means that the atomic methods (and/or blocks) can be reasoned about in a sequential manner without having to consider all possible thread interleavings. This makes it easier to reason about the behavior and correctness of these programs which simplifies code inspection, testing, and verification. It is possible to check for atomicity using techniques such as static analysis, dynamic analysis, and finite state verification.

9.2 Correctness properties of multithreaded programs

Data race freedom: A race condition occurs when two threads simultaneously access the same data variable, and at least one of the accesses is a write [7].

As a correctness property, the above is not sufficient nor necessary.

Atomicity: A method (or block) is atomic if for every arbitrarily interleaved program execution, there exists an equivalent execution with the same overall behavior where the method (or block) is executed serially [1]

Linearizability: A concurrent program is linearizable if each object is linearizable. This correctness property is non-blocking and local. [4]

Serializability: In databases, A concurrent schedule is serializable if it is equivalent to one in which transactions appear to execute using a sequential schedule. This correctness property is blocking and global. [6]

As correctness properties, the above three provide stronger guarantees than data race freedom and still allow a great deal of parallelism.

9.3 Atomizer

Atomizer is a dynamic atomicity checker that for all methods and/or blocks specified as atomic by either:

- The user as annotations
- A heuristic defined as:
  - All methods that are public or package except for main and run methods
  - All synchronized blocks

detects and reports any atomicity violations.

More specifically, the tool inputs a multithreaded Java program, generates the corresponding instrumented multithreaded Java program that uses dynamic analysis to:

- Identify race conditions based on Eraser's lockset algorithm [7]
- Check atomicity based on Lipton's reduction theory [5]

and outputs any atomicity violations with program executions that display the undesired behaviors.

9.3.1 Lockset algorithm

The Lockset algorithm is used for data race detection.

For each allocated object, for each field track to what degree it has been shared among threads:

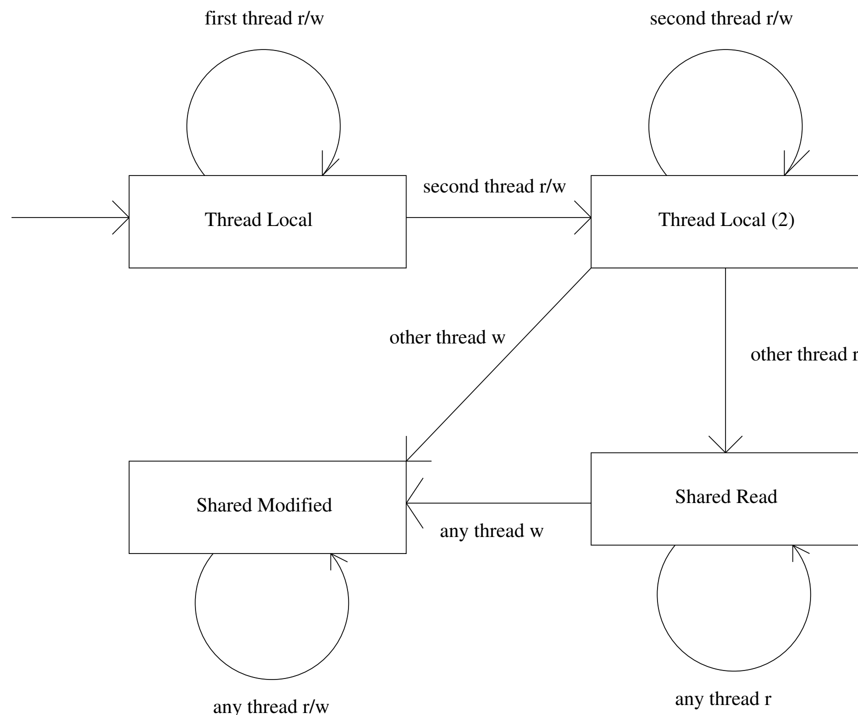

Figure 9.1: Thread sharing degree

Figure 9.1

If a field is thread local, thread local (2), or shared read, then any access is considered not racy. But if a field is shared modified then its protecting lock set must be inferred.

Version 1:

Each field has a protecting lock set inferred. If an access occurs when the lock set is empty then the access is considered racy.

For write-protected data, the above categorization is too restrictive.

Version 2:

Each field now has an access-protecting lock set and a write-protecting lock set inferred. If a read occurs when the the write-protecting lock set is empty then the read is considered racy. If a write occurs when the access-protecting lock set is empty then the write is considered racy.

9.3.2 Reduction algorithm

The Reduction algorithm is used for atomicity violation detection. It makes use of the data race information.

Version 1:

- Right-mover (R): lock acquire
- Left-momver (L): lock release
- Both-mover (B): non-racy field access
- Non-mover (N): racy field access

An atomic section must always conform to: $(R \mid B)*(N \mid L)(L \mid B)*$

For locks, the above categorization is often too restrictive. For a general lock, a lock acquire is a right-mover and a lock release is a left-move. But for a special lock as defined below its acquire and release can be considered a both-mover:

- The lock is thread local
- The lock is protected by another lock
- The lock is reentrant and this is not the first (last) acquire (release)

Version 2:

- Right-mover (R): lock acquire (not special case)
- Left-momver (L): lock release (not special case)
- Both-mover (B):
  - lock acquire (special case)
  - lock release (special case)
  - non-racy field access
- Non-mover (N): racy field access

The atomic sections can now be larger since more actions are categorized as both-movers.

9.4 Evaluation

The Atomizer was evaluated on 12 multithreaded Java programs gathered from various benchmarks.

It used the heuristic to specify methods and/or blocks as atomic. Generally, it appears that programmers often do code to this style of concurrency. The benchmarks for the most part used it.

In summary:

- There were very few false alarms (+)
- There were actual bugs found (+)
- There was a performance penalty: 1.5X - 40X (-)

9.5 Related Work

9.5.1 Static type systems

A static type system takes as input a multithreaded program where the programmer must specify:

- For each object allocated (for each field), the protecting lock(s)
- For each method and/or block, the atomicity

and outputs whether or not the atomicity is violated.

Evaluation of the static type systems:

- The programmer must provide the annotations (-)
- Since it is a static analysis, it provides better coverage and soundness guarantees then the dynamic analysis. (+)
- But is has a more restrictive domain. (-)
- In theory, the static analysis is NP-complete (-)
- In practice, they often are applicable. (+)
- The static analysis results may be used to restrict the dynamic analyses (+)

See [2].

9.5.2 Finite state verifiers

A finite state verifier takes as input a multithreaded program and perhaps the corresponding sequential program and outputs whether or not the atomicity is violated.

Evaluation of the finite state verifiers:

- The programmer may have to provide annotations (-)
- It provides better coverage and soundness guarentees then the dynamic analysis. (+)
- For integration testing, it must deal with the state explosion problem. (-)
- For unit testing, it is applicable. (+)

See [3].

References

[1] C. Flanagan and S.N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2004.

[2] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[3] J. Hatcliff, Robby, and M.B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[4] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[5] R.J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[6] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

[7] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.