This project is kindly supported by

Unit Testing for Concurrent Programs

Corky Cartwright, Mathias Ricken
{cork|mgricken}@rice.edu

Unpredictable Behavior of Concurrent Unit Tests

Unit Tests

- Written before the application code
- Must pass before code can be committed to the repository
  → Fewer bugs enter the repository
- Prevent old bugs from reappearing
  → Before a bug is fixed, a unit test exhibiting that bug is added
- Catch bugs early in development
- Current tools: JUnit, Ant

Our Experience with DrJava

- Unit tests used extensively
  → 30% of code lines are unit tests
  → 60% of code paths are tested
- Production quality software
- Constantly changing development team that includes sophomores
- Unit tests proved effective with single thread of control
- Major portion of DrJava is concurrent
  → Existing tools not effective

Existing Tools Not Effective for Concurrent Programs

- Thread switching is non-deterministic and machine-specific
  → Unit tests may pass on one run, fail on the next
- Success of unit tests does not indicate correct concurrent behavior
- Major portion of DrJava is concurrent
  → Existing tools not effective
- **Project Goal:** Develop an open-source framework for reliable, portable, automated unit testing of concurrent Java applications

Schedule-Based Unit Testing

- Execution using a fixed schedule
  → Deterministic and machine-independent execution
- Execution using all different schedules
  → Success of unit tests implies correct concurrent behavior
- Unit testing not performance critical
  → Overhead more acceptable in unit testing than in acceptance testing

Schedule Generation

- Atomic blocks
  → Delimited by synchronization points (see box on the far right)
  → Instructions in an atomic block cannot directly affect another thread if shared variables are guarded

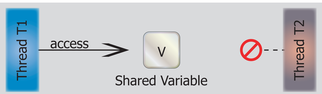


Figure: If T1 holds the lock on V, T2 cannot access V at the same time. Hence, neither thread can directly influence the other through V

- Generation Technique 1: Random delays
  → Delays inserted between atomic blocks to change scheduling behavior
  → Large number of schedules modified this way should ensure good coverage

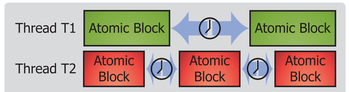


Figure: Random delays inserted between atomic blocks

- Generation Technique 2: Exhaustive search
  → Enumeration of all arrangements of atomic blocks covers all possible program behaviors
  → Fewer arrangements of atomic blocks than of instructions

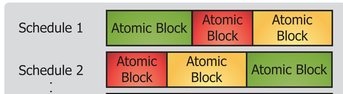


Figure: Enumerating arrangements of atomic blocks

Monitored Replay

- Custom class loader rewrites class files as they are loaded
- When the currently executed atomic block ends, one of the threads is allowed to resume, as dictated by the schedule

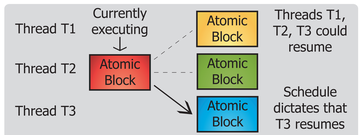


Figure: Transfer of control at the end of an atomic block

- Modified class files will run on any Java VM
  → Portable and open-source
- Bytecode can be dynamically compiled by an embedded JIT compiler
  → Faster than interpretation

Detection of Unguarded Variable Access

- Unguarded access to shared variables leads to non-deterministic behavior
  → Schedule generation relies on guarded variable access
- Custom class loader dynamically rewrites bytecode to monitor access to all shared variables
- Eraser algorithm detects if each variable is guarded by at least one lock during all accesses
- Perform cumulative intersection of the locks held during accesses to a particular variable
- If intersection is empty, variable is not properly guarded
  → No single lock held during all accesses to that particular variable

Variable X	Locks held		Variable Y	Locks held	
Access 1	A	C	Access 1	A	C
Access 2		C	Access 2		C
Access 3	B	C	Access 3	B	
Intersection		C	Intersection		

- X is guarded, intersection is non-empty
- Y is not properly guarded, intersection is empty

Figure: Cumulative intersection of locks held

Schedule Suite Validation

- Schedules depend on code base
  → May need to be regenerated if code changes
- Detect whether a code change affects synchronization behavior
  → Only regenerate schedules if necessary
- Detect whether a code change affects a unit test
  → Only execute unit test if necessary

Testing Distributed Programs

- Extend framework to distributed Java programs using RMI (Remote Method Invocation)
- Data cannot be shared except using RMI operations
  → Treat RMI operations as additional synchronization points
- Challenge: Make all machines observe the dependencies in the replayed schedule

Monitoring of Synchronization Points in Java Programs

- Java synchronization points
  → java.lang.Thread
  → Synchronized methods & blocks
  → Garbage collection & finalization
  → java.lang.System
  → Access to volatile variables
  → Remote Method Invocation (RMI)
- Monitoring necessary for schedule generation and monitored replay
  → Synchronization points delimit atomic blocks
  → Different thread may get executed when synchronization point is reached
  → Also allows for deadlock detection
- Custom class loader
  → Analyzes and modifies class files before they are loaded by the Java VM
  → Inserts calls to monitor methods before and after synchronization points

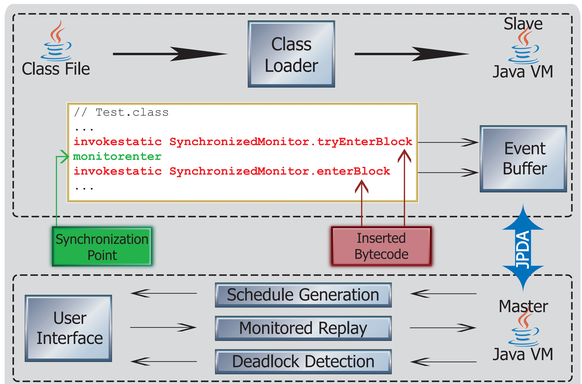
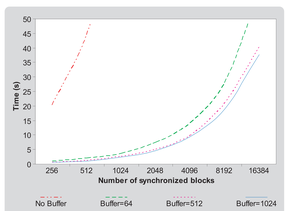


Figure: A class file is loaded by the Slave Java VM. The class loader detects synchronization points and inserts bytecode to interface with an event buffer. Using JPDA, the Master Java VM periodically reads from (schedule generation, deadlock detection) or writes to that buffer (scheduled replay).



Graph: Time in s for n synchronized blocks.

- Two virtual machines
  → Slave VM cannot crash master VM
  → Master VM cannot accidentally influence slave VM
- Buffered communication
  → Reduces communication overhead compared to naive JPDA use
  → No buffering: 256 blocks ~ 20.4 seconds
  → 1024 element buffer: 256 blocks ~ 0.5 seconds
  → New compact representation expected to be even more efficient