dBug: Systematic Testing of Distributed and Multi-Threaded Systems
Jiri Simsa, Garth Gibson, Randy Bryant

Problem Setup
- HPC computing speed grows 2X per year
- Disk bandwidth grows only 20% per year
- Random access rate grows only 7% per year
- As a result, parallel FSs grow in:
  - disks, parallelism, prefetching, delaying
- Implementing and stabilizing more complex code at HPC scales is harder each year

Background: Proving Correctness
- Model checking tools in use:
  - MaceMC, SLAM, HAVOC, Terminator, SPIN, Slayer, ...
  - No one-fits-all tool
- Typical limitations:
  - Limited range of properties / language constructs
  - Manual effort to annotate / specify / verify required
  - Proves correctness under assumptions

Background: Bug Finding
- Concrete / Symbolic execution tools in use:
  - MoDist, KLEE, eXplode, DART, VeriSoft, ...
- Execute real code in a test harness
- Typical limitations:
  - Under-constraining environment
  - Limited ability to setup test cases

Case Study: Concurrent Web Proxies
- in 15-213 students implement proxy as their final lab
- dBug was used to analyze Fall 2010 submissions
- dBug worked correctly with all 80 proxies that passed sequential checks
- dBug found concurrency errors in 25 of them
- Independent code inspection by course staff found only 5 of these errors

Case Study: Systematic Testing of Stasis
- Stasis is a flexible transactional storage system
- dBug was used to systematically enumerate ways in which Stasis unit tests can execute
- Driving Stasis unit tests through dBug resulted in 10-20x overhead
- A number of errors related to incorrect usage of shared resource was found
- As an on-going work dBug is being used to establish correctness of Stasis implementation of the ARIES protocol

dBug Approach: Design & Implementation
- Avoid annotation of source code by interposing on popular library interfaces
- Use centralized arbiter to control execution order of concurrent events
- Through arbiter scheduling, systematically explore different execution orders